PyQGIS developer cookbook

リリース *2.14*

QGIS Project

2017 年 08 月 08 日

Contents

1 はじめに **1**
- 1.1 QGIS の起動時に Python コードを実行します . . . 2
- 1.2 Python コンソール . . . 2
- 1.3 Python プラグイン . . . 3
- 1.4 Python アプリケーション . . . 3

2 プロジェクトをロード **7**

3 レイヤのロード **9**
- 3.1 ベクタレイヤ . . . 9
- 3.2 ラスタレイヤ . . . 11
- 3.3 マップレイヤレジストリ . . . 12

4 ラスターレイヤを使う **13**
- 4.1 レイヤについて . . . 13
- 4.2 レンダラ . . . 13
- 4.3 レイヤの更新 . . . 15
- 4.4 値の検索 . . . 15

5 ベクターレイヤを使う **17**
- 5.1 属性に関する情報を取得します . . . 17
- 5.2 フィーチャの選択 . . . 18
- 5.3 ベクターレイヤの反復処理 . . . 18
- 5.4 ベクターレイヤの修正 . . . 20
- 5.5 ベクターレイヤを編集バッファで修正する. . . . 21
- 5.6 空間インデックスを使う . . . 22
- 5.7 ベクターレイヤの作成 . . . 23
- 5.8 メモリープロバイダー . . . 24
- 5.9 ベクタレイヤーの外観 (シンボロジ) . . . 25
- 5.10 より詳しいトピック . . . 33

6 ジオメトリの操作 **35**
- 6.1 ジオメトリの構成 . . . 35
- 6.2 ジオメトリにアクセス . . . 36
- 6.3 ジオメトリの述語と操作 . . . 36

7 投影法サポート **39**
- 7.1 空間参照系 . . . 39
- 7.2 投影法 . . . 40

8 マップキャンバスの利用 **41**
- 8.1 マップキャンバスの埋め込み . . . 41
- 8.2 マップキャンバスでのマップツールズの利用 . . . 42
- 8.3 ラバーバンドと頂点マーカー . . . 43
- 8.4 カスタムマップツールズの書き込み . . . 44
- 8.5 カスタムマップキャンバスアイテムの書き込み . . . 45

9 地図のレンダリングと印刷 **47**
9.1 単純なレンダリング . . . 47
9.2 別の CRS とのレイヤーをレンダリング . . . 48
9.3 マップコンポーザを使った出力 . . . 48

10 表現、フィルタリング及び値の算出 **51**
10.1 パース表現 . . . 52
10.2 評価表現 . . . 52
10.3 例 . . . 53

11 設定の読み込みと保存 **55**

12 ユーザとのコミュニケーション **57**
12.1 メッセージ表示中。QgsMessageBar クラス。 . . . 57
12.2 プロセス表示中 . . . 58
12.3 ロギング . . . 59

13 Python プラグインの開発 **61**
13.1 プラグインを書く . . . 62
13.2 プラグインの内容 . . . 63
13.3 ドキュメント . . . 67
13.4 翻訳 . . . 67

14 書き込みの IDE 設定とデバッグプラグイン **69**
14.1 Windows 上で IDE を設定するメモ . . . 69
14.2 Eclipse と PyDev を利用したデバッグ . . . 70
14.3 Debugging using PDB . . . 74

15 プラグインレイヤの利用 **75**
15.1 QgsPluginLayer のサブクラス化 . . . 75

16 QGIS の旧バージョンとの互換性 **77**
16.1 プラグインメニュー . . . 77

17 プラグインをリリースする **79**
17.1 メタデータと名前 . . . 79
17.2 コードとヘルプ . . . 80
17.3 公式の python プラグインリポジトリ . . . 80

18 コードスニペット **83**
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 . . . 83
18.2 レイヤの切り替え方法 . . . 83
18.3 選択した機能の属性テーブルへのアクセス方法 . . . 84

19 処理プラグインを書く **85**
19.1 アルゴリズムプロバイダを追加するプラグインを作成する . . . 85
19.2 処理スクリプトのセットが含まれているプラグインを作成する . . . 85

20 ネットワーク分析ライブラリ **87**
20.1 一般情報 . . . 87
20.2 グラフの構築 . . . 87
20.3 グラフ分析 . . . 89

21 QGIS サーバー Python のプラグイン **95**
21.1 サーバーフィルタープラグインアーキテクチャ . . . 95
21.2 プラグインから例外を上げます . . . 97
21.3 サーバー・プラグインを書く . . . 97
21.4 アクセス制御プラグイン . . . 100

Chapter 1

はじめに

- QGIS の起動時に Python コードを実行します
  - PYQGIS_STARTUP 環境変数
  - ：ファイル：'startup.py'ファイル
- Python コンソール
- Python プラグイン
- Python アプリケーション
  - スタンドアロンスクリプトで PyQGIS を使用します
  - カスタムアプリケーションで PyQGIS を使用します
  - カスタムアプリケーションを実行する

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。

QGIS のデスクトップに Python バインディングを使用する方法、それらは次のセクションで詳しく説明されているいくつかの方法があります。

- QGIS の起動時に自動的に Python コードを実行
- 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

Python バインディングは、QGIS Server 用にも使用できます。

- 2.8 のリリースから開始し、Python のプラグインは QGIS Server 上でも利用できます（参照：'サーバーの Python プラグイン<server_plugins> '_）
- 2.11 バージョン（2015 年 8 月 11 日でマスター）から開始し、QGIS Server ライブラリは、Python アプリケーションに QGIS サーバーを埋め込むために使用することができます Python バインディングを持っています。

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。

プラグインを扱う優れたリソースは 'プラグインリポジトリ<http://plugins.qgis.org/> '_からいくつかのプラグインをダウンロードしてそのコードを調べることです。また、QGIS のインストール中の ''python /プラグイン/ ''フォルダ中にも、そのようなプラグインをどのように開発するか、最も一般的なタスクのいくつかをどのように実行するかを習得するのに使用できるプラグインが含まれています。

1.1 QGIS の起動時に Python コードを実行します

QGIS を起動するたびに Python コードを実行するには、2 つの異なる方法があります。

1.1.1 PYQGIS_STARTUP 環境変数

QGIS の初期化は、既存の Python のファイルのパスに ‘‘ PYQGIS_STARTUP‘‘環境変数を設定することで完了直前に Python コードを実行できます。

それは QGIS 内および QGIS の初期化が完了する前にこのコードを実行するための Python コードを実行するには、いくつかの方法の一つであるため、この方法では、おそらくめったに必要としません何かが、ここで言及する価値があります。この方法は、望ましくない経路を有していてもよいの sys.path を洗浄するための、またはの virt の ENV を必要とすることなく、初期環境をロードする/単離のために非常に有用であり、例えば自作または MacPorts のは、Mac にインストールされます。

1.1.2 ：ファイル：‘startup.py‘ファイル

QGIS を起動するたびに、ユーザーの Python のホームディレクトリ（通常： :file: *.qgis2 / python*）の中で :file:‘ startup.py‘という名前のファイルが検索され、そのファイルが存在する場合、埋め込まれた Python インタプリタによって実行されます。

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:

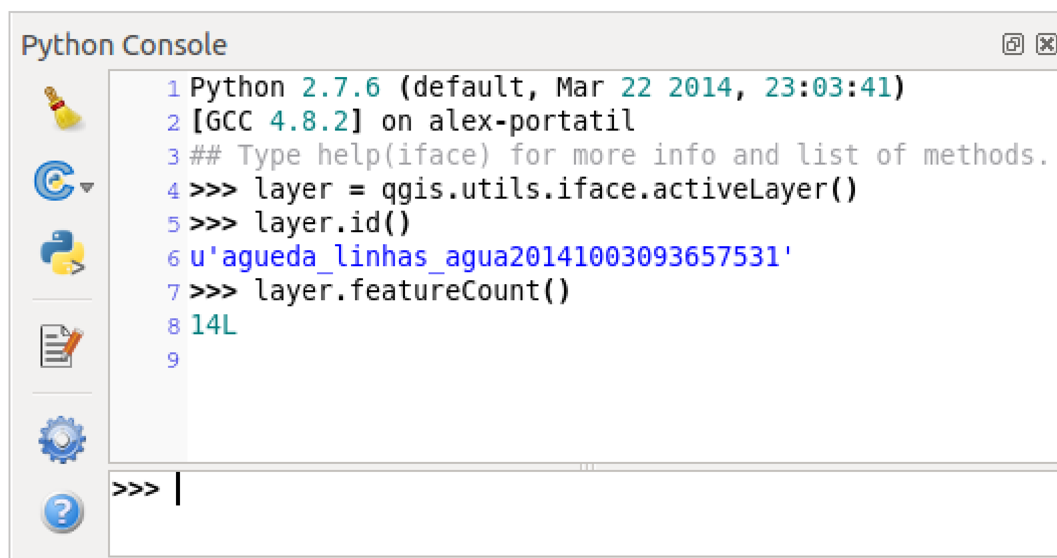

Figure 1.1: QGIS Python コンソール

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。

ノート: Python のプラグインは QGIS サーバ（*label_qgisserver*）でも利用可能です、詳細については:ref:‘server_plugins‘を参照。

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。

PyQGIS のカスタムアプリケーションまたはスタンドアロンスクリプトは、このような投影情報として QGIS のリソースを検索するように設定する必要があり、ベクトルとラスターレイヤーを読み取るためのプロバイダなど QGIS リソースは、アプリケーションやスクリプトの先頭に数行を追加することによって初期化されます。カスタムアプリケーションとスタンドアロンスクリプトの QGIS を初期化するコードが類似しているが、それぞれの例を以下に提供されます。

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。

1.4.1 スタンドアロンスクリプトで PyQGIS を使用します

スタンドアロンスクリプトを起動するには、次のコードのようなスクリプトの先頭で QGIS のリソースを初期化します。

```
from qgis.core import *

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

# create a reference to the QgsApplication, setting the
# second argument to False disables the GUI
```

```
qgs = QgsApplication([], False)

# load providers
qgs.initQgis()

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

# When your script is complete, call exitQgis() to remove the provider and
# layer registries from memory
qgs.exitQgis()
```

MOD：： *qgis.core*'**モジュールと、その後プレフィックスパスを設定する私たちは、インポートすることから始めます。プレフィックスパスは、*QGIS* がシステムにインストールされている場所です。これは、'** setPrefixPath''メソッドを呼び出すことで、スクリプトで設定されています。CONST： '**true**'を、デフォルトのパスが使用されているかどうかを制御 setPrefixPath'''の第 2 引数は、に設定されています。

QGIS のインストールパスは、プラットフォームによって異なります。と '' お使いのシステムに対するパスを見つける最も簡単な方法は、QGIS 内から:ref: '**pythonconsole'を使用し、実行している 'QgsApplication.prefixPath()'からの結果を見ることです。

プレフィックスパスを設定した後は '' QgsApplication''変数に '' qgs''への参照を保存します。スタンドアロンスクリプトを書いているので、GUI を使用する予定がないことを示すため、第二引数には '' False''を設定します。**''**QgsApplication''が設定されると、 '' qgs.initQgis（） **''**メソッドを呼び出すことで QGIS データプロバイダとレイヤーのレジストリを読み込みます。QGIS が初期化されると、残りのスクリプトを記述する準備が整いました。最後に、 '' qgs.exitQgis（） ''を呼び出して、メモリからデータプロバイダと層のレジストリを削除しておしまいです。

1.4.2 カスタムアプリケーションで PyQGIS を使用します

：REF：*standalonescript*'**とカスタム *PyQGIS* アプリケーションの唯一の違いは** ' QgsApplication''をインスタンス化する際の二番目の引数です。：定数： *true*'**を** ' False''の代わりに渡し、GUI を使用することを計画していると示します。

```
from qgis.core import *

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

# create a reference to the QgsApplication
# setting the second argument to True enables the GUI, which we need to do
# since this is a custom application
qgs = QgsApplication([], True)

# load providers
qgs.initQgis()

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

# When your script is complete, call exitQgis() to remove the provider and
# layer registries from memory
qgs.exitQgis()
```

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 :-)

1.4.3 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

プロジェクトをロード

スタンドアロンの開発時に時々、(多くの場合) プラグインから既存のプロジェクトをロードする必要がありますか (参照:は:ref: *pythonapplications*) Python アプリケーションを QGIS。

クラス:: *QgsProject* : FUNC : あなたが必要とする現在の QGIS アプリケーションにプロジェクトをロードするには、**インスタンス () 'オブジェクトを、その呼び出し :** *FUNC* **: '読んで () 'メソッドそれに渡す : クラス :** QFileInfo'オブジェクトをそれは、プロジェクトがロードされる場所からのパスが含まれています:

```
# 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'
```

FUNC : プロジェクトインスタンスの '書き込み () 'メソッドは、プロジェクトにいくつかの変更を加える (例えば、いくつかのレイヤーを追加または削除) して、変更を保存する必要がある場合は、あなたが呼び出すことができます。: FUNC : クラス :: '書き込みは () 'メソッドは、オプションの受け入れプロジェクトを保存するパスを指定することができます 'QFileInfo'を:

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

両方 : FUNC : **読んで () 'と :** *FUNC* **: '書き込み ()** funtions は、操作が成功したかどうかをチェックするために使用できるブール値を返します。

ノート: クラス :: あなたがインスタンス化する必要がキャンバスと、ロードされたプロジェクト同期させるために、QGIS スタンドアロンアプリケーションを作成している場合は、以下の例のように 'QgsLayerTreeMapCanvasBridge'を:

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

Chapter 3

レイヤのロード

- ベクタレイヤ
- ラスタレイヤ
- マップレイヤレジストリ

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

ベクトルレイヤーをロードするには、レイヤのデータソース識別子、レイヤーの名前、およびプロバイダの名前を指定します。

```
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.

QGIS でベクトルレイヤーを開き、表示する最も簡単な方法は、`QgisInterface` の addVectorLayer 関数です：

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

これは 1 つのステップで、新しいレイヤーを作成し、マップレイヤーレジストリに追加（レイヤリストに表示される）します。関数が返すのはレイヤーインスタンス、レイヤーをロードできなかった場合は ‘NONE‘です。

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 ライブラリ（シェープファイルや他の多くのファイル形式）—データソースがファイルへのパスです。
  - シェープファイルについて:

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

  - DXF 用（データソース URI 内の内部オプションに注意してください）：

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

- PostGIS のデータベース — データソースは、PostgreSQL データベースへの接続を作成するために必要なすべての情報を持つ文字列です。 `QgsDataSourceURI` クラスではあなたのためにこの文字列を生成できます。QGIS は Postgres のサポート付きでコンパイルする必要があることに注意してください、そうでない場合は、このプロバイダが使用できません。

```
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 や他の区切りのテキストファイル—区切り文字としてセミコロンを使用してファイルを開くには、x 座標およびフィールド「Y」の欄の「x」と y 座標を持つあなたはこのようなものを使用します。

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

注意：QGIS バージョン 1.7 からプロバイダ文字列を URL として構成されているので、パスの先頭に *file://* を付ける必要があります。また、それは「X」と「Y」フィールドに代わるものとして WKT（周知のテキスト）フォーマットされたジオメトリを可能にし、座標参照系を指定することを可能にします。例えば：

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

- GPX ファイル—「GPX」データプロバイダは、GPX ファイルからトラック、ルートやウェイポイントを読み込みます。ファイルを開くには、タイプ（トラック/ルート/ウェイポイント）は、URL の一部として指定する必要があります：

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

- QGIS v1.1 のサポートから SpatiaLite データベース—。クラス：同様の PostGIS データベースに ‘QgsDataSourceURI‘は、データソース識別子の生成のために使用することができます。

```
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')
```

- OGR 介して MySQL の WKB ベースジオメトリ、—データソースがテーブルへの接続文字列です。

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

- WFS 接続:. 接続は、URI と “ WFS“プロバイダを使用して定義されます。

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

URI は標準の “ urllib“ライブラリを使用して作成することができます。

```
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))
```

ノート: 既存のレイヤのデータソースを変更するには、次の例のように：クラス：'QgsVectorLayer'インスタンス上の：FUNC：'setDataSource（）'を呼び出します:

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

3.2 ラスタレイヤ

ラスターファイルへのアクセスについては、GDAL ライブラリが使用されます。これは、広い範囲のファイルフォーマットをサポートしています。何かのファイルを開くときにトラブルがある場合は、GDAL がそのフォーマットをサポートしているか確認してください（デフォルトではすべてのフォーマットが使用可能ではない)。ファイルからラスタをロードするには、そのファイル名とベース名を指定します。

```
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!"
```

クラス：'QgisInterface'同様ベクター層に、ラスタレイヤは addRasterLayer 関数を使用してロードすることができます。

```
iface.addRasterLayer("/path/to/raster/file.tif", "layer name you like")
```

これは、新しいレイヤーを作成し、1 つのステップで（それはレイヤリストに表示されて作る）マップレイヤーレジストリに追加します。

ラスターレイヤーも、WCS サービスから作成することができます。

```
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')
```

詳細な URI の設定は 'プロバイダのマニュアルに記載されています。<https://github.com/qgis/QGIS/blob/master/src/providers/wcs/URI> '_

別の方法としては、WMS サーバからラスタレイヤを読み込むことができます。しかし現在では、 API から GetCapabilities レスポンスにアクセスすることはできません—それは何をしたいレイヤーか知っている必要があります。

```
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.

レジストリにレイヤーを追加します：

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

レイヤーは終了時に自動的に破棄されますが、明示的にレイヤーを削除したい場合には次を使用します。

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

ロードされたレイヤーとレイヤー ID のリストについては、以下を使用します。

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

Chapter 4

ラスターレイヤを使う

- レイヤについて
- レンダラ
  - 単バンドラスタ
  - マルチバンドラスタ
- レイヤの更新
- 値の検索

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

ラスタレイヤは、1 つまたは複数のラスタバンドで構成されています。それらはシングルバンドまたはマルチバンドラスタと呼ばれます。 1 つのバンドは値のマトリックスを表します。通常のカラー画像（例えば、空中写真）は、赤、青、緑のバンドからなるラスタです。単一バンドレイヤーは、通常、連続変数（例えば高度）または離散変数（例えば土地利用）のいずれかを表します。場合によっては、ラスタレイヤにパレットが付属し、ラスタ値はパレットに保存されている色を参照します:

```
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 レンダラ

ラスターレイヤーがロードされると、そのタイプに基づいてデフォルトのレンダラーが取得されます。ラスターレイヤのプロパティまたはプログラムで変更できます。

現在のレンダラを照会する:

```
>>> rlayer.renderer()
<qgis._core.QgsSingleBandPseudoColorRenderer object at 0x7f471c1da8a0>
>>> rlayer.renderer().type()
u'singlebandpseudocolor'
```

レンダラを設定するためには:class:*QgsRasterLayer* の:func:*setRenderer* メソッドを使用します。利用可能なレンダラークラス（`QgsRasterRenderer` から派生）はいくつかあります：

- QgsMultiBandColorRenderer
- QgsPalettedRasterRenderer
- QgsSingleBandColorDataRenderer
- QgsSingleBandGrayRenderer
- QgsSingleBandGrayRenderer

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

ラスターレイヤー（バンドは 1 つだけと仮定）（0 から 255 までのピクセル値に対して）緑から黄までの色でレンダリングしたいとしましょう。第一段階では、 ``QgsRasterShader``オブジェクトを作成し、そのシェーダ機能を設定します：

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

シェーダは、そのカラーマップによって指定された色をマッピングします。カラーマップは、画素値とそれに関連する色を持つ項目のリストとなっています。値の補間のモードは 3 つあります。

- 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.

第二段階では、ラスタレイヤにこのシェーダを関連付けます:

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

上記のコード中の数 1 はバンド番号です（ラスタバンドは、1 からインデックス付けされている)。

4.2.2 マルチバンドラスタ

デフォルトでは、QGIS は最初の 3 つのバンドを赤、緑、青の値にマッピングしてカラー画像を作成します（これは ``MultiBandColor``描画スタイルです）場合によっては、これらの設定をオーバーライドすることもできます。 次のコードは、赤のバンド（1）と緑のバンド（2）を交換します:

```
rlayer.renderer().setGreenBand(1)
rlayer.renderer().setRedBand(2)
```

ラスタを視覚化するために必要な帯域が 1 つだけの場合は、グレーレベルまたは疑似カラーのいずれかのシングルバンド描画を選択できます。

4.3 レイヤの更新

レイヤシンボルを変更し、変更がユーザーにすぐに表示されていることを確認したい場合、これらのメソッドを呼び出します

```
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.

WMS のラスタレイヤと、これらのコマンドは動作しません。このケースでは、明示的にそれをしなければなりません

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

レイヤシンボルを変更している（それを行う方法については、ラスタとベクタレイヤーに関するセクションを参照）ケースでは、レイヤーリスト（凡例）ウィジェット内のレイヤシンボルを更新することを QGIS に強制させたい場合があります。これは以下のように行うことができます（*iface* は:class:*QgisInterface* のインスタンスです）

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

4.4 値の検索

いくつかの指定された時点で、ラスタレイヤのバンドの値のクエリを実行するには

```
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

- 属性に関する情報を取得します
- フィーチャの選択
- ベクターレイヤの反復処理
  - 属性のアクセス
  - 選択されたフィーチャへの反復処理
  - 一部のフィーチャへの反復処理
- ベクターレイヤの修正
  - フィーチャの追加
  - フィーチャの削除
  - フィーチャの修正
  - フィールドの追加または削除
- ベクターレイヤを編集バッファで修正する.
- 空間インデックスを使う
- ベクターレイヤの作成
- メモリープロバイダー
- ベクタレイヤーの外観 (シンボロジ)
  - 単一シンボルレンダラ
  - カテゴライズドシンボルレンダラ
  - 階調シンボルレンダラ
  - シンボルの操作
    * シンボルレイヤの操作
    * カスタムシンボルレイヤタイプの作成
  - カスタムレンダラの作成
- より詳しいトピック

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 属性に関する情報を取得します

:class:`QgsVectorLayer`インスタンス上:func:`pendingFields`を呼び出すことにより、ベクトルレイヤーに関連付けられたフィールドに関する情報を取得できます:

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

ノート: QGIS 2.12 から出発する、こともある:class:`QgsVectorLayer`中の:func:`fields()`もある、:func:`pendingFields`の別名である。

5.2 フィーチャの選択

QGIS デスクトップでは、地物はいろいろな方法で選択できます。ユーザーは地物をクリックしても、地図キャンバス上で長方形を描いても、表現フィルタを使用してもよいです。 選択された地物は、選択に関してユーザーの注意をひくために、通常は異なる色（デフォルトは黄色です）で強調表示されます。 プログラム的に地物を選択するとか、デフォルト色を変更すると役に立つこともあります。

選択色を変更するには、次の例に示すように、 :class:'QgsMapCanvas'の:func:'setSelectionColor() 'メソッドを使用できます:

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

所定のレイヤーのために選択した地物のリストに地物を追加するには、:func:'setSelectedFeatures()'を、地物の ID のリストをそれに渡しつつ呼び出すことができます:

```
# 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
```

別のオプションは、処理:func:'features'メソッドです：

```
import processing
features = processing.features(layer)
for feature in features:
    # do whatever you need with the feature
```

デフォルトでは、これは、選択がない場合はレイヤー中のすべての地物について、そうでない場合は選択されたすべての地物について繰り返します。 このふるまいは選択を無視するように「処理」オプションで変更できることに注意。

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
```

上記の例に示すように、空間フィルタの代わりに（あるいは加えて）属性ベースのフィルタが必要な場合は、`QgsExpression`'オブジェクトを構築し、それを`:obj:` QgsFeatureRequest'コンストラクタに渡すことができます。ここに例があります

```
# 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)
```

:class:'QgsExpression'によってサポートされる構文の詳細については、:ref:'expressions'を参照。

要求は、地物ごとに取得したデータを定義するために使用できるので、反復子はすべての地物を返しますが、それぞれの地物については部分的データを返します。

```
# 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: 属性の部分集合だけ必要な場合、または幾何情報を必要としない場合は、上の例で示されるように「QgsFeatureRequest.NoGeometry」フラグを使用するか属性の部分集合（空も可能）を指定することによって、地物要求の**速度**をかなり向上できます。

5.4 ベクターレイヤの修正

ベクトルデータプロバイダはほとんどレイヤーデータの編集をサポートします。時には彼らは、可能な編集操作の一部分だけをサポートしています。機能のどの部分がサポートされるかを見つけるために :func:`capabilities`関数を使用します

```
caps = layer.dataProvider().capabilities()
# Check if a particular capability is supported:
caps & QgsVectorDataProvider.DeleteFeatures
# Print 2 if DeleteFeatures is supported
```

利用可能なすべての機能のリストについては、`QgsVectorDataProvider の API ドキュメント <http://qgis.org/api/classQgsVectorDataProvider.html> ` _を参照してください

カンマ区切りリストでレイヤーの機能の説明テキストを印刷するには、次の例のように `capabilitiesString()` が使用できます:

```
caps_string = layer.dataProvider().capabilitiesString()
# Print:
# u'Add Features, Delete Features, Change Attribute Values,
# Add Attributes, Delete Attributes, Create Spatial Index,
# Fast Access to Features at ID, Change Geometries,
# Simplify Geometries with topological validation'
```

ベクターレイヤを編集するための以下の方法のいずれかを使用することで、変更は直接基礎となるデータストア（ファイル、データベースなど）にコミットされます。一時的な変更をしたいだけの場合は、どうするかを説明している次のセクション:ref:`編集バッファーで修正する `に跳んでください。

ノート: QGIS の内部（コンソールまたはプラグインからのいずれか）で作業している場合、ジオメトリ、スタイル、属性に行われた変更を確認するため、地図キャンバスを強制的に再描画することが必要になることもあるでしょう:

```
# 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

いくつか:class: `QgsFeature`インスタンスを作成し、プロバイダの:func:`addFeatures`メソッドにそれらのリストを渡します。これは、2 つの値を返します。結果（真/偽）および追加された地物のリスト（それらの ID はデータストアによって設定されます)。

属性を設定するには、:class: *QgsFields*`**インスタンスを渡して地物を初期化することも、追加したいフィールドの数を渡して**:func:initAttributes`を呼び出すこともできます。

```
if caps & QgsVectorDataProvider.AddFeatures:
    feat = QgsFeature(layer.pendingFields())
    feat.setAttributes([0, 'hello'])
    # Or set a single attribute by key or by index:
    feat.setAttribute('name', 'hello')
```

```
feat.setAttribute(0, 'hello')
feat.setGeometry(QgsGeometry.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: ジオメトリを変更する必要があるだけの場合は、:class:'QgsVectorLayerEditUtils'を使用することを考えてもよいでしょう。これはジオメトリを編集するための有用なメソッドをいくつか提供します（変換、移動、頂点挿入など)。

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。

レイヤーが編集モードであるかどうかを調べるには、:func:‘isEditable‘を使用します—編集機能は、編集モードがオンになっている場合にのみ動作します。編集機能の使用方法

```
# 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。

編集モードを開始するには、`startEditing()‘メソッドがあり、編集を中止するには:func:‘CommitChanges()‘と:func:‘ROLLBACK()` がありますーしかし、通常はこれらのメソッドを必要はありませんし、この機能はユーザによってトリガーされるように残します。

次の例に示すように、よりセマンティックなコードブロックにコミットとロールバックをラップする ‘‘ -:code:‘with edit(layer)‘文も使用できます：

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

これは最後には:func:‘CommitChanges()‘を自動的に呼び出します。いずれかの例外が発生した場合、それはすべての変更を:func:‘rollBack()‘します。:func:‘CommitChanges()‘中で問題に遭遇した場合（メソッドが false を返すとき）、:class:‘QgsEditError‘例外が発生します。

5.6 空間インデックスを使う

空間インデックスは、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、コードのパフォーマンスを劇的に改善します。例えば、補間アルゴリズムを書いていて、補間値の計算に使用するために与えられた位置に対して最も近い 10 点をポイントレイヤーから求める必要がある、と想像してください。空間

インデックスが無いと、QGIS がこれらの 10 点を求める方法は、すべての点から指定の場所への距離を計算してそれらの距離を比較することしかありません。これは、いくつかの場所について繰り返す必要がある場合は特に、非常に時間のかかる処理となります。もし空間インデックスがレイヤに作成されていれば、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。

空間索引は、QGIS ベクトルレイヤーのためにデフォルトで作成されていませんが、簡単に作成できます。しなければいけないことはこうです：

- 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()
```

- **クラス：インデックスに地物を追加—インデックスは ‘QgsFeature‘オブジェクトをとり、それを追加します**
  内部データ構造に。オブジェクトは手動でも作成できますし、プロバイダの:func: ‘ nextFeature()‘の前回の呼び出しからのものも使用できます。:

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

- 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 Sh

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

error = QgsVectorFileWriter.writeAsVectorFormat(layer, "my_json.json", "utf-8", None, "GeoJSON",
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 :class:`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 QgsFields object is needed
fields = QgsFields()
fields.append(QgsField("first", QVariant.Int))
fields.append(QgsField("second", QVariant.String))

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

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

# 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
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。

```
print QgsRendererV2Registry.instance().renderersList()
# Print:
[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)。

特定のベクターレイヤで使用される記号は、適切なシンボルインスタンスのインスタンスを渡しながら:func: *setSymbol()*‘**を呼び出すことによって変更できます。*ポイント* 、*ライン*、*ポリゴン*レイヤーに対するシンボルは、対応するクラス、** *:class:‘QgsMarkerSymbolV2*、`QgsLineSymbolV2`、:class:‘QgsFillSymbolV2‘の:func:‘createSimple‘関数：を呼び出すことによって作成できます。

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

たとえば、次のコード例のように、:class:‘QgsMarkerSymbolV2‘のインスタンスを渡しつつ:func: ‘setSymbol()‘を呼び出すことで、特定の**ポイント**レイヤーで使用されるシンボルを置換できます:

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

‘‘ name‘‘は、マーカーの形状を示しており、以下のいずれかとすることができます。

- 円形
- 方形
- 十字
- 長方形
- 菱形
- 五角形
- 三角形
- 正三角形
- 星形
- `regular_star`
- 矢印
- 塗りつぶし矢印
- ×印

シンボルインスタンスの最初のシンボルレイヤのプロパティの完全なリストを取得するには、サンプルコードに従うことができます:

```
print layer.rendererV2().symbol().symbolLayers()[0].properties()
# Prints
{u'angle': u'0',
u'color': u'0,128,0,255',
u'horizontal_anchor_point': u'1',
u'name': u'circle',
u'offset': u'0,0',
u'offset_map_unit_scale': u'0,0',
u'offset_unit': u'MM',
u'outline_color': u'0,0,0,255',
u'outline_style': u'solid',
u'outline_width': u'0',
u'outline_width_map_unit_scale': u'0,0',
u'outline_width_unit': u'MM',
u'scale_method': u'area',
u'size': u'2',
u'size_map_unit_scale': u'0,0',
u'size_unit': u'MM',
u'vertical_anchor_point': u'1'}
```

いくつかのプロパティを変更したい場合に便利です:

```
# You can alter a single propery...
layer.rendererV2().symbol().symbolLayer(0).setName('square')
# ... but not all properties are accessible from methods,
# you can also replace the symbol completely:
props = layer.rendererV2().symbol().symbolLayer(0).properties()
props['color'] = 'yellow'
props['name'] = 'square'
layer.rendererV2().setSymbol(QgsMarkerSymbolV2.createSimple(props))
```

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()
    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: （QgsVectorColorRampV2）色のランプでの作業（:class:‘ QgsStyleV2‘）スタイルで作業シンボルを作成/変更ルールベースのレンダラは（‘このブログ投稿<http://snorf.net/blog/2014/03/04/symbology-of-vector-layers-in-qgis-python-plugins>‘_を参照してください）探索シンボルレイヤーとレンダラ・レジストリ

Chapter 6

ジオメトリの操作

- ジオメトリの構成
- ジオメトリにアクセス
- ジオメトリの述語と操作

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。

- 座標から

```
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。

- well-known テキスト（WKT）から

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

- よく知られているバイナリ（WKB）から

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

6.2 ジオメトリにアクセス

まず、ジオメトリタイプを見つける必要があります:func:*wkbType* メソッドが使用する 1 つである—それは ‘‘QGis.WkbType‘‘列挙から値を返します。

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

代替として、‘‘QGis.GeometryType‘‘列挙から値を返す:func:*type* メソッドが使用できます。ジオメトリがマルチパートであるかどうかを調べるためのヘルパー関数:func:*isMultipart* もあります。

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.setEllipsoidalMode(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。

追加情報は次のソースで見つけることができます：

- ジオメトリ変換: Reprojected algorithm
- :class:‘QgsDistanceArea‘クラスを使用した距離と面積：‘距離行列アルゴリズム <https://raw.github.com/qgis/QGIS/release-2_0/python/plugins/processing/algs/ftools/PointDistance.py> ‘_
- **シングルパートアルゴリズムにマルチパート**<*https://raw.github.com/qgis/QGIS/release-2_0/python/plugins/processing/algs/ftools/MultipartToSingleparts.py*> _

Chapter 7

投影法サポート

- 空間参照系
- 投影法

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:

- CRS をその ID によって指定します

  ```
  # 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。

  – ：CONST：PostGIS のデータベース内で使用される *PostgisCrsId* —の ID。

  – ：CONST： *InternalCrsId* — ID が内部的に QGIS データベースで使用されます。

  – ：CONST：EPSG の組織によって割り当てられた *EpsgCrsId* —の ID

  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。

- その well-known テキスト（WKT）で CRS を指定します

  ```
  wkt = 'GEOGCS["WGS84", DATUM["WGS84", SPHEROID["WGS84", 6378137.0, 298.257223563]],'
          PRIMEM["Greenwich", 0.0], UNIT["degree",0.017453292519943295],'
          AXIS["Longitude",EAST], AXIS["Latitude",NORTH]]'
  crs = QgsCoordinateReferenceSystem(wkt)
  ```

- 無効な CRS を作成し、その後のそれを初期化する:func: 'create*'関数を使用します。。次の例では、投影法を初期化するために Proj4 文字列を使用します

  ```
  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。

空間参照系の初期化のために、QGIS ではその内部データベース:file: 'srs.db'に適切な値をルックアップする必要があることに注意。したがって、独立したアプリケーションを作成する場合、:func:'QgsApplication.setPrefixPath'で正しくパスを設定する必要があります。そうしないと、データベースの検索に失敗します。QGIS パイソンコンソールからコマンドを実行しているか、プラグインを開発している場合は、気にする必要はありません：すでにすべて設定されています。

空間参照システム情報へのアクセス

```
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

- マップキャンバスの埋め込み
- マップキャンバスでのマップツールズの利用
- ラバーバンドと頂点マーカー
- カスタムマップツールズの書き込み
- カスタムマップキャンバスアイテムの書き込み

マップキャンバスウィジェットは、おそらく QGIS 内で最も重要なウィジェットです。なぜなら、オーバーレイされた地図レイヤーで構成された地図を表示し、地図とレイヤーの間のやりとりを可能にするからです。キャンバスは、常に現在のキャンバスの範囲で定義された地図の一部を表示します。やりとりは**地図ツール**を使用して行われます：パン、ズーム、レイヤーの識別、計測、ベクトル編集などのツールがあります。他のグラフィックスプログラムと同様に、常に 1 つのツールがアクティブになっていて、使用可能なツールは切り替えできます。

マップキャンバスは:class:*QgsMapCanvas* クラスの:mod:‘ qgis.gui‘ モジュールで実装されています。実装は Qt Graphics View フレームワークに基づいています。このフレームワークは、一般に、カスタムグラフィックアイテムが配置され、ユーザがそれらと相互作用できる表面およびビューを提供します。ここではユーザがグラフィックシーン、ビュー、アイテムのコンセプトを理解するのに十分に精通していることを仮定しています。そうでない場合は、フレームワークの概要<http://qt-project.org/doc/qt-4.8/graphicsview.html> ‘_ を必ず読んでください。

地図がパン、ズームイン/アウト（または他の操作によって更新が発生）するたびに、地図は現在の範囲内で再び投影されます。レイヤーは画像に投影され（:class: ‘QgsMapRenderer‘クラスを使用して）、その画像がキャンバスに表示されます。マップを表示するグラフィックアイテム（Qt グラフィックスビューフレームワークに関して）は、:class: ‘QgsMapCanvasMap‘クラスです。このクラスは、投影された地図の更新も制御します。バックグラウンドとして機能するこのアイテムの他にも**マップキャンバスアイテム**はありえます。典型的なマップキャンバスアイテムは、ラバーバンド（計測、ベクトル編集などに使用）や頂点マーカーです。キャンバスアイテムは、通常、新しいポリゴンを作成するときなど、マップツールの視覚的フィードバックを表示するために使用されます。マップツールは、ポリゴンの現在の形状を示すラバーバンドキャンバスアイテムを作成します。すべてのマップキャンバスアイテムは:class: ‘QgsMapCanvasItem‘のサブクラスで、基本的な ‘‘QGraphicsItem‘‘オブジェクトにいくつかの機能を追加します。

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

マップキャンバスは他の Qt ウィジェットと同じようにウィジェットなので、使い方は簡単です

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

これは、マップキャンバスを備えたスタンドアロンのウィンドウを生成します。既存のウィジェットまたはウィンドウに埋め込むこともできます。 .ui ファイルと Qt Designer を使用する場合は、フォーム上に ‘‘ QWidget‘‘を置いてクラス名として ‘‘ QgsMapCanvas‘‘を設定し、ヘッダファイルとして ‘‘ qgis.gui‘‘を設定して新しいクラスにします。 ‘‘ pyuic4‘‘ユーティリティーがそれを処理します。これは、キャンバスを埋め込む非常に便利な方法です。もう 1 つの可能性は、マップキャンバスや他のウィジェット（メインウィンドウやダイアログの子として）を構築し、レイアウトを作成するためのコードを手作業で書くことです。

デフォルトでは、マップキャンバスの背景色は黒でありアンチエイリアスは使用されません。背景を白に設定し、投影をなめらかにするためのアンチエイリアスを有効にするには

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

（不思議に思われる場合は ‘‘Qt‘‘は *PyQt4.QtCore‘‘* **モジュールから来ていると** *Qt.white‘‘* **は、事前に定義された** *‘‘QColor‘‘* **インスタンスの一つである。**、）

今度は地図レイヤーをいくつか追加します。最初にレイヤーを開き、地図レイヤーレジストリに追加します。次に、キャンバスの範囲を設定し、キャンバスのレイヤーのリストを設定します

```
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)])
```

これらのコマンドを実行した後、キャンバスには読み込んだレイヤーが表示されているはずです。

8.2 マップキャンバスでのマップツールズの利用

次の例では、マップキャンバスと地図のパンとズームのための基本的なマップツールを含むウィンドウを作成します。各ツールを起動するためのアクションが作成されます。パンは:class: *QgsMapToolPan* で行います。クラスは:class:‘ QgsMapToolZoom‘ のインスタンスでズームイン/ズームアウトします。アクションはチェック可能に設定され、アクションのチェック/非チェック状態を自動的に処理（マップツールがアクティブになると、そのアクションに選択済みのマークが付き、以前のマップツールのアクションが選択解除される）できるように後ほどツールに割り当てられます。マップツールは:func: *setMapTool* メソッドを使って起動されます。

```
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)
```

上記のコードはファイル、例えば:file: ‘mywnd.py‘に入れ、QGIS 内の Python コンソールで試すことができます。このコードは、現在選択されているレイヤーを新しく作成されたキャンバスに配置します

```
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

キャンバス内の地図にさらにデータを表示するには、マップキャンバスアイテムを使用します。カスタムのキャンバスアイテムクラスを作成することができます（以下で説明します）。便利な 2 つの便利なキャンバスアイテムクラスがあります。ポリラインやポリゴンを描画するための:class: *QgsRubberBand* 、描画ポイントのための:class:‘ QgsVertexMarker‘ です。それらは両方ともマップ座標で動作するため、キャンバスがパンまたはズームされるときにはシェイプは自動的に移動/拡大縮小されます。

ポリラインを表示するには

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

ポリゴンを表示するには

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

ポリゴンの点が普通のリストではないことに注意してください。実際には、多角形の線状の環を含有する環のリストです：最初のリングは外側の境界であり、さらに（オプションの）環はポリゴンの穴に対応します。

ラバーバンドはいくらかカスタマイズできます、すなわち、その色と線幅を変更することがが可能です

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

キャンバスアイテムはキャンバスシーンにバインドされています。それらを一時的に隠す（そして再表示する）には、`hide`'と`:func:()` show'のコンボを使います。アイテムを完全に削除するには、キャンバスのシーンから削除する必要があります

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

（C ++ではアイテムを削除することだけ可能ですが、Python では `` del r``は参照を削除するだけでありオブジェクトはキャンバスの所有物なのでそのまま残ります）

ラバーバンドはポイントを描画するためにも使用できますが、これには:class: *QgsVertexMarker*'**クラスの方が適しています（***:class:* QgsRubberBand'だと目的のポイントの周りに矩形を描くだけになるでしょう）。頂点マーカーの使い方は

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

これは [0,0] の位置に赤い十字を描きます。アイコンの種類、サイズ、色、ペンの幅はカスタマイズできます

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

一時的な頂点マーカーの隠蔽とキャンバスからそれらを除去するために、ラバーバンドと同じことが適用されます。

8.4 カスタムマップツールズの書き込み

自分でカスタムツールを書くことで、キャンバス上でユーザーが実行するアクションにカスタム動作を実装できます。

地図ツールは:class:*QgsMapTool*'**クラスまたは派生クラスから継承し、既に見たように***:func:* setMapTool'メソッドを使用してキャンバス内のアクティブなツールとして選択する必要があります。

キャンバスをクリックしてドラッグすることで矩形範囲を定義できるマップツールの例を次に示します。矩形が定義されると、境界座標がコンソールに表示されます。前述のラバーバンド要素を使用して、選択されている矩形が定義されていることを示します。

```
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):
        super(RectangleMapTool, self).deactivate()
        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: マップキャンバスアイテムを作成する方法

```
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

- 単純なレンダリング
- 別の CRS とのレイヤーをレンダリング
- マップコンポーザを使った出力
  - ラスタイメージへの出力
  - PDF への出力

どちらかの簡単な方法使用してそれを実行します：あり入力データがマップとしてレンダリングされるべき 2 つのアプローチが一般的クラス： *QgsMapRenderer*‘**またはでマップを構成することにより、より多くの微調整出力を生成します。***class***：** QgsComposition‘クラスや友人を。

9.1 単純なレンダリング

クラス：使用して、いくつかの層をレンダリング— *QgsMapRenderer*‘**を宛先ペイントデバイス（** QImage‘‘、‘‘ QPainter‘‘等）を作成し、レイヤーセット、範囲、出力サイズを設定してレンダリングを行います

```
# 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 = QgsRectangle(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 別の CRS とのレイヤーをレンダリング

複数のレイヤーがあってそれらが異なる CRS を持っている場合、上記の単純な例はおそらく動作しません：範囲計算から正しい値を取得するには、明示的に先の CRS を設定し、以下の例のように OTF の再投影を有効にする必要があります（レンダラの設定部分が報告されるのみ）

```
...
# 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

地図コンポーザーは、上に示した簡単なレンダリングよりも洗練された出力をしたいと思った場合に、非常に便利なツールです。コンポーザーを使用すると、地図ビュー、ラベル、凡例、テーブルと通常は紙の地図上に存在する他の要素からなる複雑なマップレイアウトを作成することが可能です。レイアウトは、その後 PDF、ラスタ画像にエクスポートしたり、直接プリンタに印刷できます。

コンポーザーは、クラスの束で構成されています。彼らはすべてのコアライブラリに属しています。それは GUI ライブラリでは利用できないですが、QGIS アプリケーションは、要素の配置のための便利な GUI を持っています。'Qt のグラフィックスビューフレームワーク<http://doc.qt.io/qt-4.8/qgraphicsview.html> ' _に精通していない方は、今、ドキュメントをチェックすることをお勧めします、作曲はそれに基づいているため。'QGraphicView の実装の Python ドキュメント<http://pyqt.sourceforge.net/Docs/PyQt4/qgraphicsview.html> _'もチェックしてください。

コンポーザーの中心クラスは:class:*QGraphicsScene*'**から派生する***:class:* QgsComposition'です。1 つ作成してみましょう

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

作品は:class: 'QgsMapRenderer'のインスタンスを取ることに注意してください。このコードでは QGIS アプリケーション内で実行されること、よってマップキャンバスから地図レンダラーを使用していることを期待しています。作品は、地図レンダラーからの各種パラメータ、最も重要な地図レイヤーのデフォルトセットと現在の範囲を使用します。スタンドアロンアプリケーションでコンポーザーを使用するときは、上記のセクションに示すように、同じように、独自の地図レンダラーインスタンスを作成して、それを作品に渡すことができます。

:class:'QgsComposerItem'クラスには、これらの要素は、の子孫でなければならない—作品に様々な要素（マップ、ラベルを、...）を追加することも可能です。現在サポートされている項目は、次のとおりです。

- 地図—このアイテムは地図自体を置くためのライブラリを伝えます。ここでは、地図を作成し、全体の用紙サイズの上に伸ばします

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

- ラベル—ラベルを表示できます。そのフォント、色、配置及びマージンを変更することが可能です

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

- 凡例

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

- スケールバー

```
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

デフォルトでは、新しく作成された作図項目の位置はゼロ（ページの左上隅）でサイズはゼロです。位置とサイズは常にミリメートルで測定されます

```
# 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)
```

フレームは、デフォルトでは、各項目を中心に描かれています。フレームを削除する方法

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

手で作図アイテムを作成するほか、QGIS は基本的に（XML 構文を持つ）.qpt ファイルに保存されたすべての項目を持つ作品である作図テンプレートをサポートしています。残念ながら、この機能は API ではまだ利用できません。

作品の準備ができたら（作図項目が作成され、作品に追加されたら）、ラスタおよび/またはベクトル出力を生成するために進むことができます。

作品のためのデフォルトの出力設定は、ページサイズ A4 および解像度 300 DPI です。必要に応じてそれらを変更できます。用紙サイズは、ミリメートル単位で指定されます

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

9.3.1 ラスタイメージへの出力

次のコード断片は、ラスタイメージに組成物をレンダリングする方法を示して

```
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

次のコードは、PDF ファイルに作品をレンダリングします

```
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

- パース表現
- 評価表現
  - 基本表現
  - 地物に関わる表現
  - エラー処理
- 例

QGIS では、SQL 風の式の構文解析のためのいくつかサポートしています。SQL 構文の小さなサブセットのみがサポートされています。式は、ブール述語（True または False を返す）として、または関数（スカラー値を返す）のどちらかとして評価できます。使用可能な関数の完全なリストについては、ユーザーマニュアル中の *vector_expressions* を参照。

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`
- 変換関数：‘‘ to_int‘‘、‘‘ to_real‘‘、‘‘ to_string‘‘、‘‘ to_date‘‘
- ジオメトリファンクション: `$area`, `$length`
- ジオメトリ処理関数：‘‘ $ x‘‘、‘‘$の y‘‘、‘‘ $の geometry‘‘、‘‘ num_geometries‘‘、‘‘ centroid‘‘

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。

これらの変数は保存され、Qt と QGIS API の助けを借りて取得できます。各変数について、変数にアクセスするために使用されるキーを選択する必要があります—ユーザの好みの色のためにキー「favourite_color」またはその他の意味のある文字列を使用できます。キーの名前をつけるときは何らかの構造を持たせることをお勧めします。

我々は、異なる種類の設定をすることができます：

- **グローバル設定は** —彼らは、特定のマシンのユーザにバインドされています。QGIS 自体がグローバル設定の多くを保存し、例えば、メインウィンドウのサイズまたはデフォルトでは許容範囲をスナップ。クラス :: *QSettings*'**クラスこの機能は、によって** *Qt* **フレームワークによって直接提供されます。デフォルトでは、（***Mac OS X* **の）（***Windows* **の場合）—レジストリ、の***.plist* **ファイルまたは（***Unix* **上）***.ini* **ファイルで保存する設定のシステムの「ネイティブ」な方法でこのクラスを格納設定。**'*QSettings* **のドキュメント**<*http://doc.qt.io/qt-4.8/qsettings.html*> _包括的であるので、我々は単純な例を提供します

  ```
  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.

- **プロジェクト設定** —異なるプロジェクト間で変動し、したがって、それらは、プロジェクトファイルに接続されています。地図キャンバスの背景色または先座標参照系（CRS）の例である黄色の背景と UTM の突起が別のもののために優れていながら—白背景と WGS84 は、一つのプロジェクトに適しているかもしれません。使い方の例は次のとおり

  ```
  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。

- **マップレイヤの設定** —これらの設定は、プロジェクトでのマップレイヤの特定のインスタンスに関連しています。彼らは、* *層の基礎となるデータソースに接続されていないので、あなたは 1 つのシェープファイルの 2 つのマップレイヤのインスタンスを作成した場合、彼らは設定を共有することはありません。設定はプロジェクトファイルに保存されているので、ユーザは、プロジェクトを再度開いた場合、層関連の設定が再び存在します。この機能は、QGIS v1.4 以降に追加されました。API は、それが取る— QSettings に似ており、QVariant インスタンスを返します。

```
# 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

- メッセージ表示中。QgsMessageBar クラス。
- プロセス表示中
- ロギング

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。

```
from qgis.gui import QgsMessageBar
iface.messageBar().pushMessage("Error", "I’m sorry Dave, I’m afraid I can’t do that", level=QgsMes
```

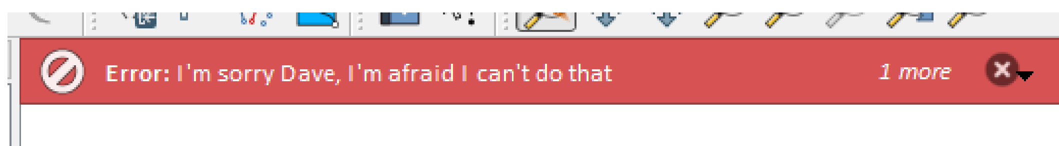

Figure 12.1: QGIS メッセージバー

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=Q
```

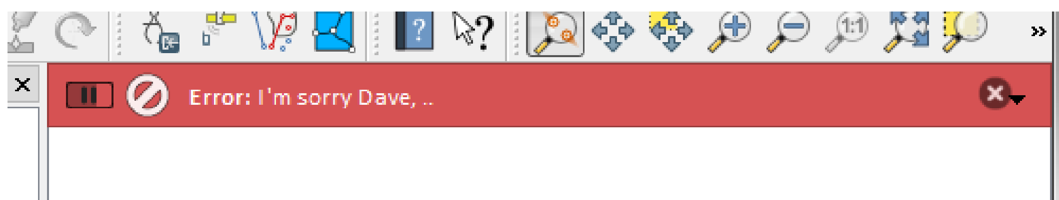

Figure 12.2: ターマー付き QGIS メッセージバー

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。

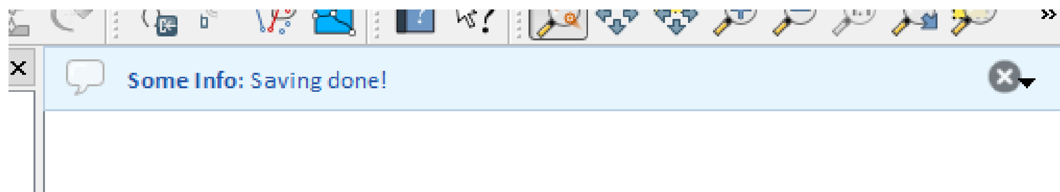

Figure 12.3: QGIS メッセージバー (お知らせ)

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

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)
```

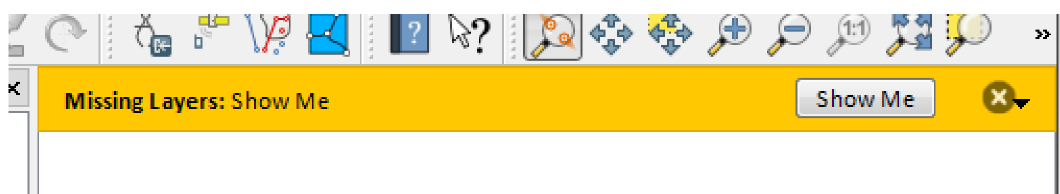

Figure 12.4: ボタン付きの QGIS メッセージバー

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。

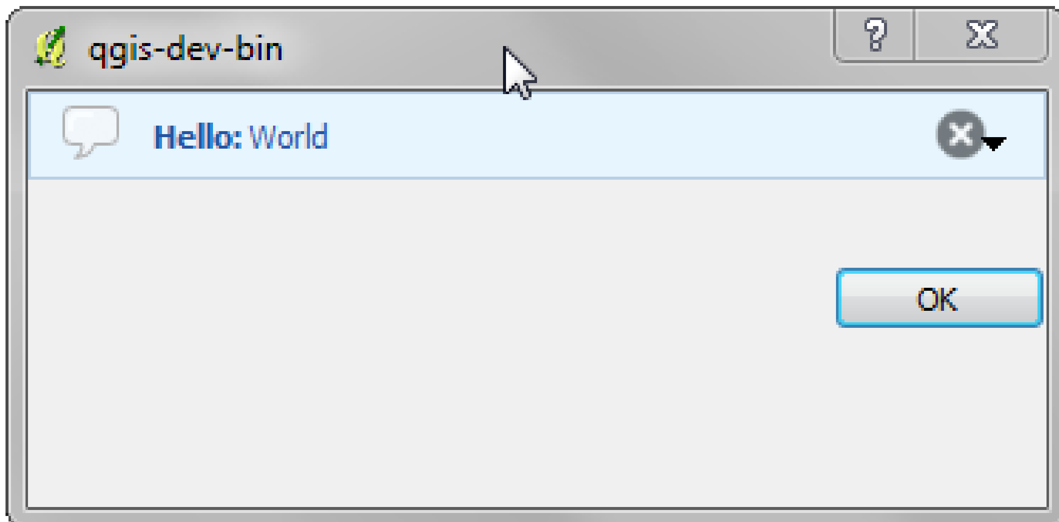

Figure 12.5: カスタムダイアログ内の QGIS メッセージバー

```
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)
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

- プラグインを書く
  - プラグインファイル
- プラグインの内容
  - プラグインメタデータ
  - __init__.py
  - mainPlugin.py
  - リソースファイル
- ドキュメント
- 翻訳
  - 必要なソフトウェア
  - ファイルとディレクトリ
    - .PRO ファイル
    - .ts ファイル
    - .qm ファイル
  - プラグインの読み込み

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 のプラグインは QGIS プラグインマネージャで、C++プラグインと一緒にリストされています。それらはこれらのパスの中から検索されています:

- UNIX/Mac: `~/.qgis2/python/plugins` および `(qgis_prefix)/share/qgis/python/plugins`
- Windows: `~/.qgis2/python/plugins` および `(qgis_prefix)/python/plugins`

Windows でのホームディレクトリ (上の ~ で示される) は通常 `C:\Documents and Settings\(user)` (Windows XP またはそれ以前) や `C:\Users\(user)` のようなものです。QGIS は Python 2.7 を使用しているのでこれらのパスのサブディレクトリは、プラグインとしてインポートできる Python パッケージと見なされるには __init__.py を含む必要があります。

ノート: 既存のディレクトリのパスを *QGIS_PLUGINPATH* にセットすることによって、このパスをプラグインを検索するパスのリストに追加することができます。

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. **ファイルの作成**: 次に説明するファイルを作成します。開始点は (`__init__.py`)。**プラグインメタデータ** (`metadata.txt`) に記入します。Python プラグインの本体 (`mainplugin.py`)。QT デザイナのフォーム (`form.ui`) とその `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. **発行**: あなたのプラグインを QGIS リポジトリに発行するか、個人的な”GIS の武器” の” 武器庫” として独自のリポジトリを作ります。

13.1 プラグインを書く

QGIS に Python プラグインが導入されてから、いくつものプラグインが Plugin Repositories wiki page に登場しています。あなたはそれらのいくつかを見つけることができますし、PyQGIS のプログラミングを学ぶためにソースを利用できます。また、開発の努力が重複していないか調べることができます。QGIS チームは **公式の** *python* **プラグインリポジトリ** を保持しています。プラグインを作成する準備はできたけれど何をすべきかについてのアイデアがありませんか? Python Plugin Ideas wiki page にはコミュニティからの願いが並べられています!

13.1.1 プラグインファイル

これらはサンプルプラグインのディレクトリ構造です

```
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` = プラグインの開始点。 `classFactory()` メソッドを持つ必要があります。その他の初期化コードを持つこともできます。
- `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` = Qt デザイナによって作成された.xml ドキュメント。フォームのリソースへの相対パスが含まれています。
- `resources.py` = 上記の.qrc ファイルが Python に変換されたもの。
- `form.ui` = Qt デザイナで作成された GUI
- `form.py` = 上記の form.ui が Python に変換されたもの。
- `metadata.txt` = QGIS >= 1.8.0 で必要です。全般情報やバージョン、名前、そしてプラグインのウェブサイトやインフラストラクチャによって使用されるいくつかのメタデータが含まれます。QGIS 2.0 以降では `__init__.py` からのメタデータは受け付けられず、 `metadata.txt` が必要です。

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:** あなたのプラグインを **公式の** *python* **プラグインリポジトリ** へアップロードするつもりなら、プラグインがプラグイン **検証** で要求されるいくつかの追加の規則に従っていることをチェックする必要があります

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	True	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	作者のメールアドレスで、 ユーザがログインしなければ QGIS プラグインマネージャや website で表示されることはありません、つまり他のプラグイン作者やプラグイン website の管理者のみが見ることができます
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	コンマ区切りのリスト。個々のタグの内部にスペースを入れることは可能です。
homepage	False	プラグインのホームページを指す有効な URL
repository	True	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	web に親和性の高い画像 (PNG, JPEG) のファイル名または相対パス (プラグインの圧縮されたパッケージのベースフォルダに対する)
category	False	*Raster*, *Vector*, *Database*, *Web* のいずれか

デフォルトではプラグインは **プラグイン** メニューに配置されます (次のセクションでプラグインのメニューエントリを追加する方法を見ます) が **ラスタ** や **ベクタ**, **データベース**, *Web* メニューに配置することもできます。

"category" のメタデータエントリはそれを明記するためにあります。プラグインはそれに応じて分類することができます。このメタデータエントリはユーザーのためのヒントとして使用され、どこに (どのメニューに) プラグインがあるかを示しています。"category" として指定できる値は Vector, Raster, Database, Web です。たとえば、あなたのプラグインが **ラスタ** メニューから利用できる場合は、 これを `metadata.txt` に追加します

```
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。

metadata.txt の例

```
; 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
tracker=http://bugs.itopen.it
repository=http://www.itopen.it/repo
; end of mandatory metadata

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

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

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

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

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

; if empty, it will be automatically set to major version + .99
qgisMaximumVersion=2.0
```

13.2.2 `__init__.py`

このファイルは Python のインポートシステムに必要とされます。QGIS にプラグインが読み込まれる時に呼ばれる `classFactory()` 関数がこのファイルに含まれている必要があります。それは `QgisInterface` のインスタンスへの参照を受け取って、`mainplugin.py` のプラグインクラスのインスタンスを返す必要があります — 私たちの場合はそれは `TestPlugin` という名前のクラスです (下記参照)。`__init__.py` はこんな感じに見えるべきです:

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

## any other initialisation needed
```

13.2.3 mainPlugin.py

これが魔法が起きるところで、魔法はこのように見えます: (例えば 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!"
```

メインのプラグインのソースファイル (例. mainPlugin.py) に含まれる必要があるプラグイン用の関数は:

- `__init__` –> QGIS のインターフェイスとのアクセスを与えます
- `initGui()` –> プラグインがロードされたときに呼び出されます
- `unload()` –> プラグインがアンロードされたときに呼び出されます

上記の例では `addPluginToMenu()` が使用されています。これは対応するメニューアクションを *Plugins* メニューに追加します。別のメニューにアクションを追加する他の方法も存在します。それらのメソッドの一覧です:

- `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()
```

カスタマイズを可能にするために QAction と QMenu の ‘‘objectName‘‘ をプラグイン固有の名前に設定することを忘れないで下さい。

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>
```

他のプラグインや QGIS のいずれかの部分と衝突しない接頭辞を使用するのが良いです。そうでなければあなたが望んでいないリソースを得るかもしれません。そして、リソースを格納する Python ファイルを生成する必要があります。 **pyrcc4** コマンドで行います:

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

ノート: Windows 環境では、**pyrcc4** をコマンドプロンプトまたは Powershell から実行しようとするとたぶん "Windows cannot access the specified device, path, or file [...]" というエラーを受け取るでしょう。簡単な解決法はたぶん OSGeo4W Shell を使うことですが、あなたがもし PATH 環境変数を書き換えたり実行ファイルのパスをはっきりと指定するのが苦でなければ <Your QGIS Install Directory>\bin\pyrcc4.exe で見つける事ができます。

これで以上です... なにも複雑なものはありません :)

すべてうまくいったらプラグインマネージャであなたのプラグインを見つけてロードすることができるはずです。そしてツールバーアイコンや適切なメニューアイテムが選択された時にコンソールにメッセージが表示されるはずです。

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。

`showPluginHelp()` 関数は呼び出し元のモジュールと同じディレクトリでヘルプファイルを探します。`index-ll_cc.html`, `index-ll.html`, `index-en.html`, `index-en_us.html`, `index.html` の順に探し、はじめに見つけたものを表示します。ここで `ll_cc` は QGIS のロケールです。ドキュメントの複数の翻訳をプラグインに含めることができます。

`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。

13.4 翻訳

いくつかのステップでプラグインの翻訳をするための環境がセットアップでき、あなたのコンピュータの言語設定に依存していたプラグインが他の言語環境でも読むことができるでしょう。

13.4.1 必要なソフトウェア

翻訳ファイルを作成及び管理するもっとも簡単な方法は Qt Linguist をインストールすることです。Linux のような環境では次をタイプすることでインストールできます:

```
sudo apt-get install qt4-dev-tools
```

13.4.2 ファイルとディレクトリ

プラグインを作成したときにプラグインのメインのディレクトリに `i18n` フォルダがあるのを見つけることができます。

全ての翻訳ファイルはこのディレクトリにあります。

.PRO ファイル

まず、‘‘ .pro‘‘ファイルを作成する必要があり、それは QtLinguist によって管理できる*プロジェクト*ファイルです。

この `.pro` はあなたが翻訳したい全てのファイルとフォームを含めなければいけません。このファイルは翻訳ファイルや変数をセットアップするのに使われます。 pro ファイルの例です:

```
FORMS = ../ui/*

SOURCES = ../your_plugin.py

TRANSLATIONS = your_plugin_it.ts
```

今回のケースでは全ての UI は `../ui` に配置され、あなたはこれらの全てを翻訳しようとしています。

さらに、 `your_plugin.py` ファイルは QGIS ツールバーにあるあなたのプラグインの全てのメニューとサブメニューから **呼び出され** 、これらも全て翻訳しようとしています。

最後に *TRANSLATIONS* 変数で翻訳したい言語を特定します。

> **警告:** `ts` ファイルの名前は `your_plugin_` + 言語 + `.ts` となるように命名するよう気をつけてください、そうでなければ言語の読み込みに失敗するでしょう！2 文字の短縮形の言語を使います (**it** はイタリア語、 **de** はドイツ語、などなど...)

.ts ファイル

一度 `.pro` を作ったら、あなたのプラグインの言語 (ごと) の `.ts` ファイル (たち) を生成する準備ができました。

ターミナルを開いて、 `your_plugin/i18n` に移動して次のように入力します:

```
lupdate your_plugin.pro
```

`your_plugin_language.ts` ファイル (ら) がみつかるはずです。

`.ts` を **Qt Linguist** で開いて翻訳を開始します。

.qm ファイル

あなたのプラグインの翻訳が終わったら (もしいくつかの文字列が終わっていなかったらソースの言語の文字列が使われるでしょう) `.qm` ファイルを作る必要があります (`.ts` ファイルをコンパイルしたもので、QGIS で使われます)。

ターミナルを開いて `your_plugin/i18n` ディレクトリに cd して、次のように入力します:

```
lrelease your_plugin.ts
```

これで、 `i18n` ディレクトリに `your_plugin.qm` ファイル (たち) が見つかるでしょう。

13.4.3 プラグインの読み込み

QGIS を開いてあなたのプラグインの翻訳を確認するには、言語を変更して (*Settings* → *Options* → *Language*) QGIS を再起動します。

これで的確な言語であなたのプラグインを見ることができます。

> **警告:** プラグインで何かを変更した場合（新しい UI に、新しいメニュー、など。）** ‘‘ .ts‘‘と ‘‘ .qm‘‘ファイルの両方の**更新バージョンを再度生成する必要がありますので、上記のコマンド再度実行します。

Chapter 14

書き込みのIDE設定とデバッグプラグイン

- Windows 上で IDE を設定するメモ
- Eclipse と PyDev を利用したデバッグ
  - インストール
  - QGIS の準備
  - Eclipse のセットアップ
  - デバッガの設定
  - eclipse での API の理解
- Debugging using PDB

各プログラマが自分の好みの IDE /テキストエディタがありますが、ここでの書き込みのために人気の IDE のを設定し、QGIS Python のプラグインをデバッグするためのいくつかの推奨事項があります。

14.1 Windows 上で IDE を設定するメモ

Linux ではプラグインを開発するために必要な追加の構成はありません。しかし、Windows 上では、同じ環境設定を持ち、QGIS と同じライブラリとインタプリタを使用することを確認する必要があります。これを行うための最速の方法は、QGIS の起動バッチファイルを変更することです。

OSGeo4W インストーラを使用した場合は OSGeo4W インストールの ‘‘ bin‘‘フォルダの下にこれを見つけることができます。ファイル :: ‘C : \ OSGeo4W \ビン\ QGIS-unstable.bat‘のようなものを探してください。

‘Pyscripter IDE <http://code.google.com/p/pyscripter> ‘ _を使用するために、しなければならないことは：

- :file: ‘QGIS-unstable.bat‘のコピーを作成し、名前を ‘‘pyscripter.bat‘‘に変更します。
- エディタで開きます。そして、最後の行、QGIS を起動するもの、を削除します。
- Pyscripter 実行ファイルを指す行を追加し、（> = 2.0 QGIS の場合は 2.7）を使用する Python のバージョンを設定し、コマンドライン引数を追加します
- さらに、QGIS が使用する Python の DLL を Pyscripter が見つけられるフォルダを指す引数を追加します。これは OSGeoW インストールの bin フォルダの下に見つけることができます

```
@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
```

これで、このバッチファイルをダブルクリックすれば、正しいパスで Pyscripter が開始するようになります。

開発者の中では Eclipse が Pyscripter よりも人気が高く、よく選択されます。以下のセクションでは、プラグインの開発とテストのためのプラグインの設定方法について説明します。 Windows で Eclipse を使用する環境を準備するには、バッチファイルを作成して Eclipse を起動する必要があります。

そのバッチファイルを作成するには、次の手順を実行します。

- :file: *qgis_core.dll*'**が常駐するフォルダを検索します。通常、これは** *:file:* C:\ OSGeo4W \apps\ QGIS \ bin'ですが、自身の QGIS アプリケーションをコンパイルした場合、これは :file:'output/bin/ RelWithDebInfo'中のビルトフォルダの中にあります
- :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.x

14.2.2 QGIS の準備

QGIS 自体で行われるいくつかの準備があります。** ** **リモートデバッグと**プラグインのリロード機能：二つのプラグインが注目されています。

- menuselection： **に行くプラグイン** - > **管理やプラグインをインストールします**...
- *リモートデバッグ*を検索するには、(現時点では、それはまだ実験的ですので、下の実験的なプラグインを有効：guilabel： 'それが表示されない場合は Options'タブ)。それをインストールしてください。
- 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

Eclipse で、新しいプロジェクトを作成します。*一般的なプロジェクト*を選択しておいて本当のソースを後でリンクできるので、このプロジェクトをどこに配置するかは実際は問題になりません。

今、新しいプロジェクトを右クリックし、：menuselection： '新規 - > フォルダ 'を選択します。

** [詳細] **をクリックし、guilabel：'別の場所に リンク（リンクフォルダ）'を選択します。すでにデバッグしたいソースがある場合はこれらを選択してください。そうでない場合は、すでに説明したようにフォルダを作成してください。

今度はビュー：guilabel： 'プロジェクト Explorer'で、ソースツリーがポップアップし、コードで作業を開始できます。すでに利用可能な構文の強調表示や他のすべての強力な IDE ツールが使用できます。

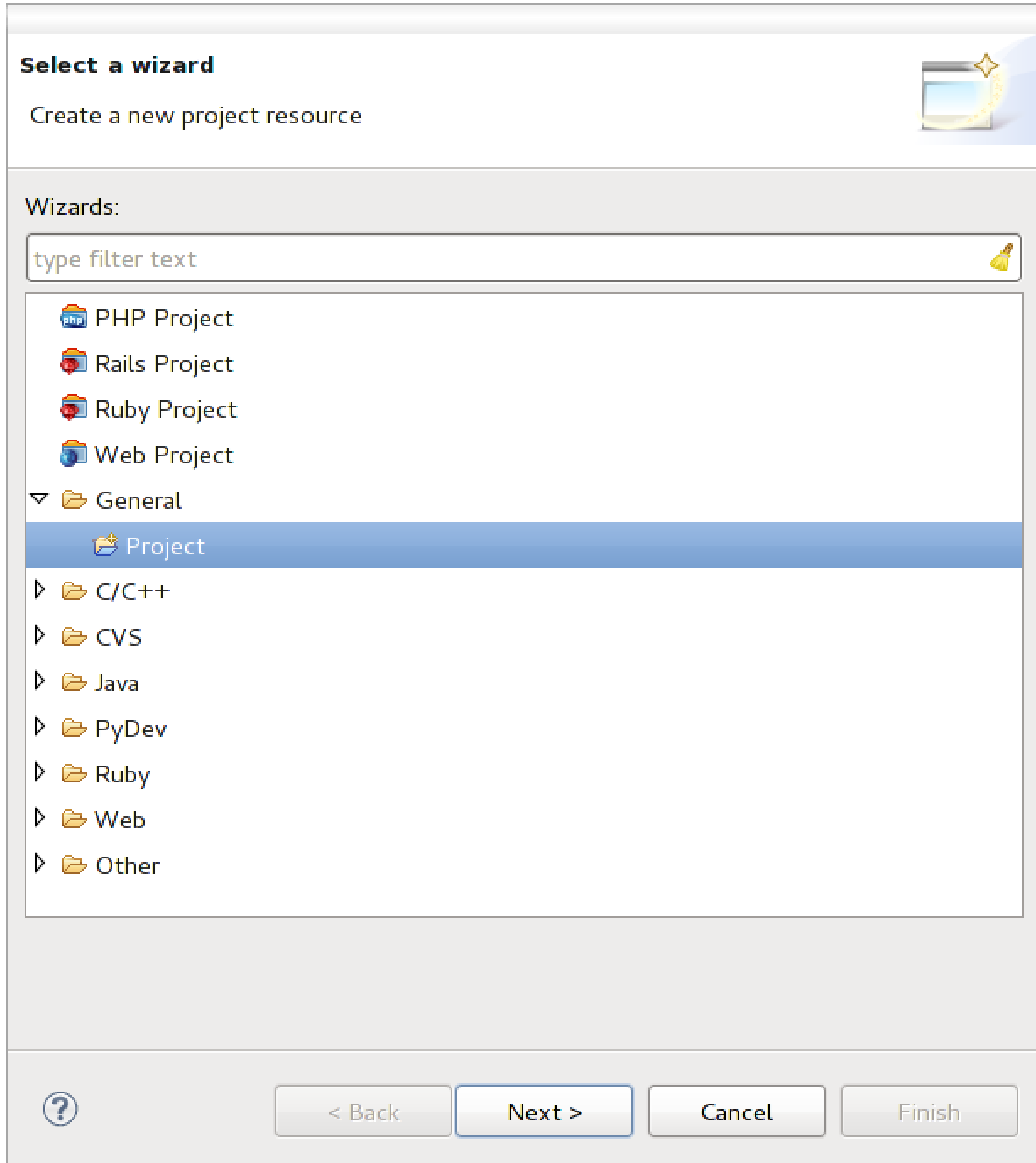

Figure 14.1: Eclipse プロジェクト

14.2.4 デバッガの設定

デバッガの作業を取得するには、Eclipse でデバッグパースペクティブに切り替えます（：menuselection：**ウィンドウ** - >**パースペクティブを開く** - >**その他** - > *DEBUG*)。

menuselection： - >デバッグの開始 Server‘ ‘PyDev はを今すぐ選択することにより、PyDev はデバッグサーバを起動します。

Eclipse は QGIS からデバッグサーバーへの接続を待っています。QGIS がデバッグサーバーに接続すると、Python スクリプトを制御できます。それはまさに私たちが* Remote Debug *プラグインをインストールしたものです。だからまだ起動していなければ QGIS を起動し、バグのシンボルをクリックしてください。

今、ブレークポイントを設定でき、コードがそこに達すると実行が停止し、プラグインの現在の状態を検査できます。（ブレークポイントは下の画像の緑色の点で、ブレークポイントを設定したい行の左空白でダブルクリックすることで設定します)。

```
 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

今利用できる非常に興味深いものはデバッグコンソールです。先に進む前に、現在実行がブレークポイントで停止していることを確認してください。

（ - > view‘表示 ‘ウィンドウ :: menuselection）コンソールビューを開きます。guilabel： ‘デバッグ Server‘コンソールは非常に興味深いものではありませんそれは表示されます。しかし、もっと面白い PyDev はデバッグコンソールに変更できますボタン** [開くコンソール] **があります。** [コンソールを開く]**ボタンの隣にある矢印をクリックして、* PyDev コンソール*を選択してください。ウィンドウが表示され、どのコンソールを起動したいかを尋ねてきます。* PyDev デバッグコンソール*を選択してください。グレーアウトしており、デバッガを起動して有効なフレームを選択するよう言われた場合は、リモートデバッガが付属され現在ブレークポイント上にいることを確かめてください　説明します。

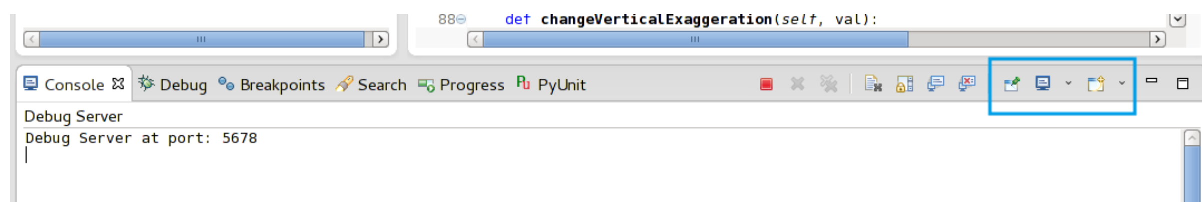

Figure 14.3: PyDev デバッグコンソール

今はもうインタラクティブコンソールが使用できるので、現在のコンテキスト内から任意のコマンドをテストできます。変数を操作するとか、API 呼び出しするとか、何でも好きにできます。

ちょっと面倒なのですが、コマンドを入力するたびコンソールは Debug Server に戻ります。この動作を停止するには、Debug Server ページで* Pin Console *ボタンをクリックしますが、少なくとも現在のデバッグセッションではこの決定は記憶されます。

14.2.5 eclipse での API の理解

非常に便利な機能は、Eclipse が実際に QGIS の API についてわかっているようにすることです。これにより、タイプミスがないかコードを確認できます。これだけでなく、Eclipse でのインポートから API 呼び出しへ自動入力する支援を可能にします。

これを行うため、Eclipse では QGIS ライブラリファイルを解析し、そこにすべての情報を取得します。しなければならないことは、どこのライブラリを検索するかを Eclipse に伝えることだけです。

クリック：menuselection：‘ウィンドウ - >設定 - > PyDev は - >通訳 - > Python‘を。

ウィンドウの上部と下部にいくつかのタブで設定済みの Python インタプリタ（現在 QGIS のための python2.7）が表示されます。私たちにとって興味深いタブは、* ライブラリー*および *強制ビルトイン*です。

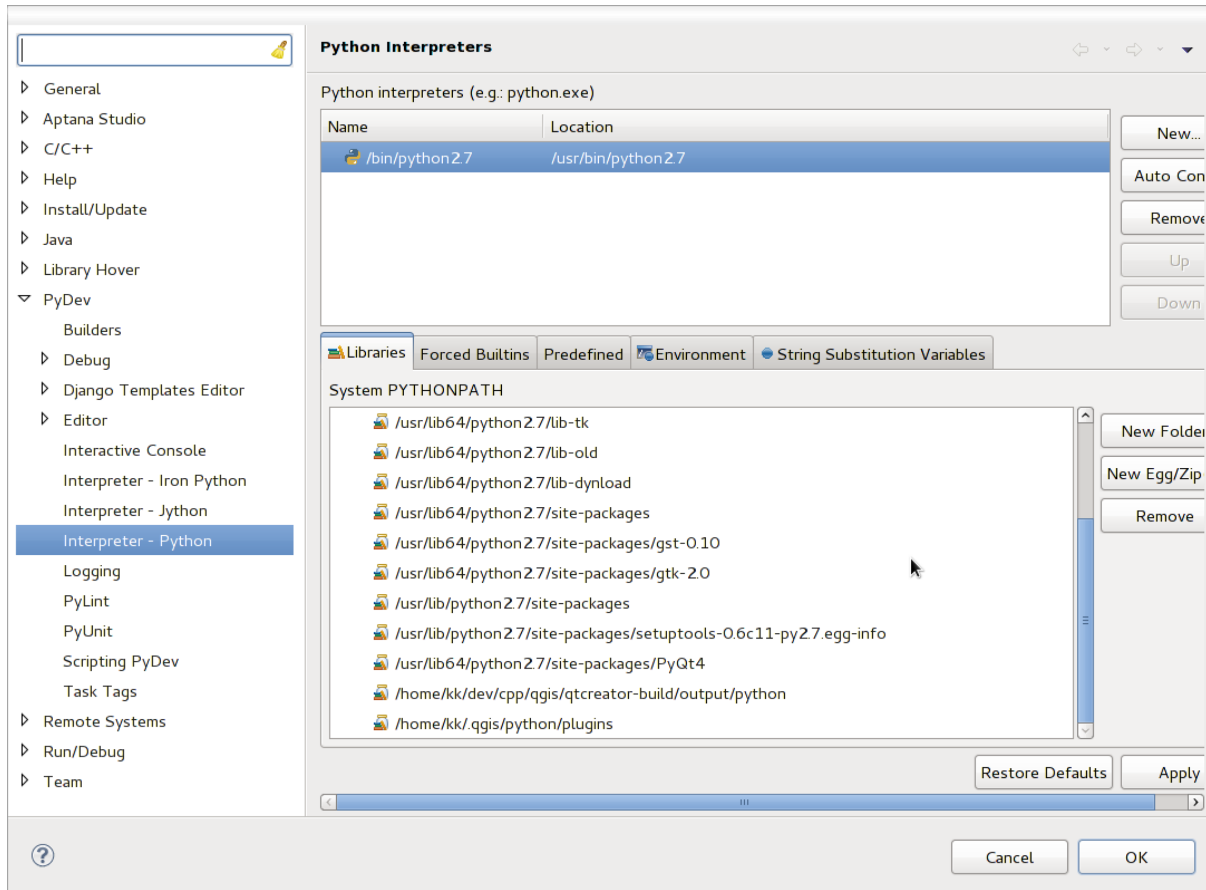

Figure 14.4: PyDev デバッグコンソール

まずライブラリ] タブを開きます。新規フォルダを追加し、QGIS のインストールの Python のフォルダを選択します。このフォルダがどこにあるか（それはプラグインフォルダではありません）オープン QGIS がわからない場合は、Python のコンソールを起動し、簡単に入力します ‘‘ qgis‘‘を入力してを押します。それは使用するモジュールとそのパスを QGIS たことが表示されます。末尾の ‘‘ / QGIS このパスから/ __の init __。pyc‘‘をストリップと、あなたが探しているパスを持っています。

プラグインフォルダもここに追加する必要があります（Linux の場合：file： *~/.qgis2/python/plugins*）。

次に*強制組込関数*タブにジャンプし、*新規... *をクリックして ‘‘ qgis‘‘と入力してください。これで Eclipse が QGIS の API を解析するようになります。おそらく Eclipse が PyQt4 の API もわかるようにしたいでしょう。そのためには強制組込関数として PyQt4 も追加します。それはおそらく、すでにライブラリタブに存在しなければなりません。

*OK*をクリックし、完了します。

ノート： QGIS API が変更されるたびに（たとえば、QGIS マスターをコンパイルして SIP ファイルを変更

した場合)、このページに戻って*適用*をクリックする必要があります。これにより、Eclipse はすべてのライブラリを再び解析できます。

エクリプスの別の可能な設定のために **このリンクをチェックし、***QGIS Python* **のプラグインで動作するように、**<*http://linfiniti.com/2011/12/remote-debugging-qgis-python-plugins-with-pydev*> _

14.3 Debugging using PDB

Eclipse などの IDE を使用しない場合は、次の手順に従って PDB を使用してデバッグできます。

まず、デバッグしたい場所にこのコードを追加します

```
# 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。

Linux 上で実行する:

```
$ ./Qgis
```

Mac OS X は実行:

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

アプリケーションがブレークポイントに到達したとき、コンソールで入力できます！

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

以下は最小限の QgsPluginLayer 実装の例です。これは Watermark example plugin の抜粋です

```
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

（：guilabel：*Raster*、：guilabel：‘Vector‘、：guilabel：*Database*‘か：*guilabel*：web‘は次）新しいメニューの一つに、プラグインのメニュー項目を配置する場合、あなたはのコードを変更する必要があります：FUNC：‘initGui（）‘と：FUNC：‘アンロード（）‘関数。これらの新しいメニューのみ QGIS 2.0 で利用可能と大きいので、最初のステップは実行 QGIS バージョンは、すべての必要な機能を有していることを確認することです。guilabel：‘Plugins‘メニューに新しいメニューが用意されていた場合、我々はそうでない場合、我々は古いを使用しますが、このメニューの下に私たちのプラグインを配置します。guilabel：ここでの例です‘Raster‘メニュー

```
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

プラグインをリリースする

- メタデータと名前
- コードとヘルプ
- 公式の python プラグインリポジトリ
  - 許可
  - 運用の信託
  - 検証
  - プラグイン構造

プラグインの準備ができ、そのプラグインが誰かのために役立つことあると思ったら、躊躇わずに:ref: ‘official_pyqgis_repository‘にアップロードしてください。そのページでは、プラグインのインストーラでうまく動作するプラグインを準備する方法についてパッケージ化のガイドラインも見つかります。あるいは、独自のプラグインのリポジトリを設定したい場合は、プラグインとそのメタデータの一覧を表示する単純な XML ファイルを作成してください。例は他の ‘プラグインリポジトリ<http://www.qgis.org/wiki/Python_Plugin_Repositories> ‘_を参照してください。

次の提案に特別な注意してください：

17.1 メタデータと名前

- 既存のプラグインとあまりにも類似した名前を使用しないよう
- プラグインが既存のプラグインと類似の機能を持っている場合は、「このプラグインについて」フィールドに違いを説明し、ユーザーがインストールして試さなくてもどれを使用するべきかわかるようにしてください
- プラグイン自体の名前に「プラグイン」を繰り返さないよう
- より詳細な手順については、1 行の説明のためのメタデータでは約フィールドを説明フィールドを使用します
- コードリポジトリ、バグトラッカー、およびホーム・ページが含まれます。これは非常にコラボレーションの可能性を向上させます、そして利用可能な Web インフラストラクチャの 1 つ（GitHub の、GitLab、の Bitbucket など）で非常に簡単に行うことができます
- 注意してタグを選択します（例：ベクトル）情報価値のないものを避け、すでに他のユーザーによって使用されるものを好む（プラグインの Web サイトを参照してください）
- 適切なアイコンを追加、デフォルトの 1 のままにしないでください。使用するスタイルの提案のための QGIS インターフェースを参照してください

17.2 コードとヘルプ

- リポジトリに（例えば.gitignore）生成されたファイル（UI _ *。PY、resources_rc.py、生成されたヘルプファイル...）と役に立たないものを含みません
- 適切なメニューにプラグインを追加（ベクター、ラスター、ウェブ、データベース）
- 適切な場合（解析を実行するプラグイン）、処理フレームワークの subplugin としてプラグインを追加することを検討：これは、ユーザーがバッチでそれを実行できるようになり、より複雑なワークフローでそれを統合するために、インターフェイスを設計する負担からあなたを解放します
- 最低限のドキュメンテーションと、テストと理解に役立つ場合はサンプルデータを含めてください。

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: ガバナンスのドキュメントへのリンクを挿入

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。
- 特別な権限 ‘plugins.can_approve‘を持つユーザーがアップロードしたバージョンは自動的に承認されます
- 特別な権限 ‘plugins.can_approve‘を持つユーザーは、彼らがプラグイン*所有者*のリスト中にある限り、他人によってアップロードされたバージョンを承認できます
- 特定のプラグインは *スタッフ*ユーザまたはプラグイン *所有者*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

スタッフはフロントエンドアプリケーションを介して ‘plugins.can_approve‘許可を設定し、選択したプラグインの作成者に*信頼*を付与することができます。

プラグインの詳細ビューには、プラグインの作成者やプラグイン*所有者*への信頼を付与するために直接リンクを提供しています。

17.3.3 検証

プラグインのメタデータを自動的にインポートし、プラグインがアップロードされたときに圧縮されたパッケージから検証されます。

公式リポジトリ上にプラグインをアップロードするときに注意するべき検証規則がいくつかあります：

1. プラグインを含むメインフォルダの名前は、ASCII 文字（A-Z および a-z）、数字、アンダースコア（_）とマイナス（ - ）しか含んではならず、また数字で始めることはできません。

2. `metadata.txt` が必要です
3. REF：に記載されているすべての必要なメタデータ ‘メタデータテーブル 1‘存在している必要があります
4. ‘VERSION‘メタデータフィールドは一意である必要があります

17.3.4 プラグイン構造

検証に続いて、プラグインの圧縮（.zip ファイル）パッケージは、機能プラグインとして検証する特定の構造を持たなければならないルール。プラグインは、プラグインフォルダをユーザーの内部で解凍されるように、それは他のプラグインに干渉しないように.zip ファイル内の独自のディレクトリです持っている必要があります。必須ファイルは、次のとおりです：ファイル： *metadata.txt*‘**と：ファイル：** __init __ py‘。（：ファイル： *resources.qrc*）プラグインを表現するために ‘ README‘そしてもちろんアイコン：ファイル：しかし、持っていいだろう。以下は plugin.zip がどのように見えるべきかの一例です。

```
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

- 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
- レイヤの切り替え方法
- 選択した機能の属性テーブルへのアクセス方法

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

処理プラグインを書く

- アルゴリズムプロバイダを追加するプラグインを作成する
- 処理スクリプトのセットが含まれているプラグインを作成する

開発しようとしているプラグインの種類によっては、処理アルゴリズム（またはそれらのセット）として機能を追加する方が良い場合もあるでしょう。そうすれば、QGIS 内でのより良い統合がなされ（これは、モデラーやバッチ処理インターフェースといった、「処理」のコンポーネントの中で実行できますので）、追加の機能、そして迅速な開発時間です（「処理」は作業時間の大部分を占めるので）。

この文書では、処理アルゴリズムとして機能を追加する新しいプラグインを作成する方法について説明します。

おもなメカニズムは２つ

- アルゴリズムプロバイダを追加するプラグインの作成：このオプションは、より複雑ですが、より多くの柔軟性を提供します
- 処理スクリプトのセットが含まれているプラグインの作成：最も簡単な解決策を、あなただけの処理スクリプトファイルのセットを必要とします。

19.1 アルゴリズムプロバイダを追加するプラグインを作成する

アルゴリズムプロバイダを作成するには、次の手順を実行します。

- プラグイン Builder プラグインをインストールします。
- プラグインビルダーを使用して新しいプラグインを作成します。プラグインビルダーがテンプレートを使用するように求めてきたら、「処理プロバイダ」を選択します。
- 作成したプラグインには、単一のアルゴリズムを持つプロバイダが含まれます。プロバイダファイルおよびアルゴリズムファイルの両方とも、完全にコメントされ、プロバイダを修正したりさらにアルゴリズムを追加する方法についての情報が含まれています。詳細については、それらを参照してください。

19.2 処理スクリプトのセットが含まれているプラグインを作成する

処理スクリプトのセットを作成するには、次の手順を実行します。

- PyQGIS 料理本で説明したようにスクリプトを作成します。追加したいすべてのスクリプトは、処理ツールボックス中でそれらが利用可能である必要があります。

- 処理ツールボックスの* Scripts / Tools **グループで、** Create script collection plugin *項目をダブルクリックします。プラグインに追加するスクリプト（ツールボックスの使用可能なセットから選択）と、プラグインのメタデータに必要な追加情報を選択するウィンドウが表示されます。
- OK をクリックすると、プラグインが作成されます。
- スクリプト python ファイルを結果のプラグインフォルダの* scripts *フォルダに追加することにより、プラグインに追加のスクリプトを追加できます。

Chapter 20

ネットワーク分析ライブラリ

- 一般情報
- グラフの構築
- グラフ分析
  - 最短経路を見つける
  - 利用可能領域

改訂 ‘ee19294562 <https://github.com/qgis/QGIS/commit/ee19294562b00c6ce957945f14c1727210cffdf7> ‘_（QGIS> = 1.8）から始め、新しいネットワーク解析ライブラリが QGIS コア解析ライブラリに追加されました。ライブラリ：

- 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。
- グラフ理論からの基本的なメソッドを実装します（現在はダイクストラ法のみ）

ネットワーク解析ライブラリは RoadGraph コアプラグインからの基本的機能をエクスポートすることによって作成されました。今はそのメソッドをプラグインで、または Python のコンソールから直接使用できます。

20.1 一般情報

手短に言えば、一般的なユースケースは、次のように記述できます。

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）

20.2 グラフの構築

最初にする必要がある事は—入力データを準備することです、つまりベクターレイヤーをグラフに変換することです。これからのすべての操作は、レイヤーではなく、このグラフを使用します。

ソースとしてどんなポリラインベクトルレイヤーも使用できます。ポリラインの頂点は、グラフの頂点となり、ポリラインのセグメントは、グラフの辺です。いくつかのノードが同じ座標を持っている場合、それらは同じグラフの頂点です。だから共通のノードを持つ 2 つの線は接続しています。

さらに、グラフの作成時には、入力ベクトルレイヤーに好きな数だけ追加の点を「固定」する（「結びつける」）ことが可能です。追加の点それぞれについて、対応箇所—最も近いグラフの頂点または最も近いグラフの辺、が探し出されます。後者の場合、辺は分割されて新しい頂点が追加されるでしょう。

ベクターレイヤ属性と辺の長さは、辺の性質として使用できます。

ベクターレイヤーからグラフへの変換は **'ビルダー<http://en.wikipedia.org/wiki/Builder_pattern>'_** プログラミングパターンを使用して行われます。グラフは、いわゆるディレクターを用いて構成されています。今のところ唯一のディレクターがあります： **'QgsLineVectorLayerDirector<http://qgis.org/api/classQgsLineVectorLayerDirector.html>'_** 。ディレクターは、グラフを作成するためにビルダーによって使用される線ベクトルレイヤーからグラフを構築するために使用される基本的な設定を設定します。現在、ディレクターとの場合のように、ビルダーは一つだけ存在します： **'QgsGraphBuilder<http://qgis.org/api/classQgsGraphBuilder.html>'_** 、それは 'QgsGraph <http://qgis.org/api/classQgsGraph.html>'_ オブジェクトを作成します。自作のビルダーは 'BGL<http://www.boost.org/doc/libs/1_48_0/libs/graph/doc/index.html>'_ か、NetworkX などのライブラリと互換性のあるグラフを構築するように実装できます。

辺性質を計算するにはプログラミングパターン **'戦略<http://en.wikipedia.org/wiki/Strategy_pattern>'_** が使用されます。今のところ利用できるのは、経路の長さを考慮に入れる **'QgsDistanceArcProperter<http://qgis.org/api/classQgsDistanceArcProperter.html>'_** 戦略だけです。すべての必要なパラメータを使用する独自の戦略を実装できます。例えば、RoadGraph プラグインは、属性から辺長と速度値を使用して旅行時間を計算する戦略を使用します。

プロセスに飛び込む時間です。

まず第一に、このライブラリを使用するために、networkanalysis モジュールをインポートする必要があります

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

ディレクターを作成するためのその後いくつかの例

```
# 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)
```

ディレクタを構築するために、ベクターレイヤーを渡さなければなりません。これはグラフ構造および各道路セグメント上で許される移動（一方向または双方向の動き、順または逆の方向）についての情報のソースとして使用されるでしょう。呼び出しは次のようになります

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

そして、ここでこれらのパラメータは何を意味するかの完全なリストは以下のとおりです。

- `vl` —グラフを構築するために使用される ベクターレイヤー
- `directionFieldId` —道路の方向に関する情報が格納されている属性テーブルのフィールドのインデックス。`` -1``、その後、まったくこの情報を使用しない場合。整数。
- `directDirectionValue` —順方向（線の最初の点から最後の点へ移動）の道に対するフィールド値。文字列。
- `reverseDirectionValue` —逆方向（線の最後の点から最初の点へ移動）の道に対するフィールド値。文字列。
- `bothDirectionValue` —双方向道路に対するフィールド値（例えば最初の点から最後まで、また最後から最初まで移動できる道に対する）。文字列。
- `defaultDirection` —デフォルトの道路の方向。この値は、フィールド `directionFieldId```が設定されていない、または上で指定された 3 つの値のいずれとも

異なる値を有する道路に使用されるであろう。整数。``1 は順方向、 2 は逆方向、 3 は両方向を示します。

それから、辺性質を計算するための戦略を作成することが必要です

```
properter = QgsDistanceArcProperter()
```

そして、ディレクターにこの戦略について教えます

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

これで、グラフを作成するビルダーを使用できます。QgsGraphBuilder クラスのコンストラクタには、いくつかの引数を取ります。

- crs —使用する空間参照系。必須の引数です。
- otfEnabled — "その場で" 再投影を使用するかどうか。デフォルトでは `True` (OTF を使用).
- 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)
```

分析に使用されるいくつかのポイントを、定義することもできます。例えば

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

これですべてが整いましたので、グラフを構築し、それにこれらの点を「結びつける」ことができます。

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

グラフを構築するには（レイヤー中の図形数とレイヤーのサイズのに応じて）いくらか時間がかかることがあります。 `tiedPoints` は「結ばれた」点の座標とのリストです。ビルド操作が完了すると、グラフは取得して分析のために使用できます

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

次のコードで、点の頂点インデックスを取得できます

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

20.3 グラフ分析

ネットワーク分析はこの二つの質問に対する答えを見つけるために使用されます：どの頂点が接続されているか、どのように最短経路を検索するか。これらの問題を解決するため、ネットワーク解析ライブラリではダイクストラのアルゴリズムを提供しています。

ダイクストラ法は、グラフの１つの頂点から他のすべての頂点への最短ルートおよび最適化パラメータの値を見つけます。結果は、最短経路木として表現できます。

最短経路木は、次の性質を有する有向重み付きグラフ（より正確には、木）です。

- 流入する辺がない頂点が 1 つだけあります - 木の根
- 他のすべての頂点には流入する辺が１つだけあります
- 頂点 B が頂点 A から到達可能である場合、このグラフ上の A から B への経路は、単一の利用可能な経路であり、それは最適（最短）です

最短経路木を得るには、 **'QgsGraphAnalyzer<http://qgis.org/api/classQgsGraphAnalyzer.html>'_** クラスの `shortestTree()` と `dijkstra()` メソッドを使用してください。より速く動作し、より多くのメモリを効率的に使用するため、 メソッド `dijkstra()` を使用することをお勧めします。

最短経路木の周りを歩くしたい場合 `shortestTree()` メソッドが便利です。それは、常に新しいグラフオブジェクト（QgsGraph）を作成し、三つの変数を受けとります。

- 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)
```

`dijkstra()` メソッドは引数は同じですが 2 つの配列を返します。流入辺が存在しない場合は、最初の配列要素 i に流入辺のインデックスまたは-1 を含有します。二番目の配列要素 i には木の根から頂点 i までの距離が入ります。頂点 i が根から到達不能である場合は DOUBLE_MAX が入ります。

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

`shortestTree()` メソッド（TOC でラインストリングのレイヤーを選択し、自分で座標を置き換える）ここで作成したグラフを使用して最短パスツリーを表示するには、いくつかの非常に単純なコードです。 **警告** ：このコードはほんの一例として使用してください、それは多くの **‘QgsRubberBand<http://qgis.org/api/classQgsRubberBand.html>‘_** オブジェクトを作成し、大規模なデータセットでは低速になることがあります。

```
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
```

同じですが `dijkstra()` メソッドを使用して

```
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())
```

20.3.1 最短経路を見つける

2 点間の最適経路を見つけるために下のアプローチが使用されます。両方の点（始点 A および終点 B）は構築されたときにグラフに「結ばれます」。それから、メソッド `shortestTree()` または `dijkstra()` を使用して、開始点 A に根を持つ最短経路木を構築します。また、同じ木の中に終点 B を見つけ、木を点 A から点 B まで歩き始めます。全体のアルゴリズムは次のように書けます

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

この時点において、この経路で走行中に訪問される頂点の反転リストの形（頂点は逆順で終点から始点へと列挙されている）で、経路が得られます。

これはメソッド `shortestTree()` を使用する QGIS Python コンソールのためのサンプルコードです（TOC 中のラインストリングレイヤーを選択して、コードの座標を自分のものに置き換える必要があります）

```
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)
```

そしてこれは同じサンプルですが `dijkstra()` メソッドを使用しています

```
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)
```

20.3.2 利用可能領域

頂点 A に対する利用可能領域とは、頂点 A から到達可能であり、A からこれらの頂点までの経路のコストがある値以下になるような、グラフの頂点の部分集合です。

より明確に、これは次の例で示すことができる。「消防署があります。市内の部分へ、消防車が 5 分で、10 分で、15 分で到達できますか？」。これらの質問への回答は、消防署の利用可能領域です。

利用可能領域を見つけるためには、 `QgsGraphAnalyzer` クラスのメソッド `dijkstra()` を使用できます。事前に定義された値とコスト配列の要素を比較すれば十分です。コスト [i] が所定値以下の場合は、頂点 i は利用可能領域内に、そうでない場合は利用可能領域外にあります。

より難しい問題は、利用可能領域の境界を取得することです。下限はまだ到達できる頂点の集合であり、上限は到達できない頂点の集合です。実際にはこれは簡単です：それは、最短経路木の辺に基づいて利用可能境界された辺の元頂点はアクセス可能であり、辺の先頂点ではありません。

例があります

```
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))
```

Chapter 21

QGIS サーバー Python のプラグイン

- サーバーフィルタープラグインアーキテクチャ
  - requestReady
  - sendResponse
  - responseComplete
- プラグインから例外を上げげます
- サーバー・プラグインを書く
  - プラグインファイル
  - __init__.py
  - HelloServer.py
  - 入力を変更します
  - 出力の変更または交換
- アクセス制御プラグイン
  - プラグインファイル
  - __init__.py
  - AccessControl.py
  - 層のフィルタ式
  - layerFilterSubsetString
  - layerPermissions
  - authorizedLayerAttributes
  - allowToEdit
  - cacheKey

Python のプラグインも QGIS Server 上で実行することができます（参照：は：ref： *label_qgisserver*）:（：クラス：‘ QgsServerInterface‘）*サーバインタフェース*を使用してサーバー上で実行されている Python のプラグインは、既存のコアサービスの動作を変更することができます（ ** WMS 、 WFS **など）。

（：クラス： *QgsServerFilter*）*サーバフィルタインタフェース*で、私たちは、生成された出力を変更したりしても、新たなサービスを提供することで、入力パラメータを変更することができます。

（：クラス： *QgsAccessControlFilter*）*アクセス制御インタフェース*で、リクエストごとにいくつかのアクセス制限を適用できます。

21.1 サーバーフィルタープラグインアーキテクチャ

FCGI アプリケーションの起動時にサーバーの python プラグインは、一度ロードされます。クラス :: 彼らは、1 つまたは複数の登録 ‘QgsServerFilter‘を（この時点から、あなたは ‘サーバーのプラグイン API ドキュメントに役立つ簡単に見て見つけるかもしれない<http://qgis.org/api/group__server.html> ‘_）。各フィルタは、少なくとも 3 つのコールバックを実装する必要があります。

- `requestReady()`

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

（：クラス： *QgsRequestHandler*）すべてのフィルタは、リクエスト/レスポンスオブジェクトへのアクセス権を持っており、そのすべてのプロパティ（入力/出力）を操作し、（以下で見るように非常に特定の方法で）例外を発生させることができます。

ここでは、フィルタのコールバックが呼ばれ、一般的なサーバーのセッションとを示す擬似コードは次のとおりです。

- **着信要求を取得**
  - GET / POST / SOAP リクエストハンドラを作成
  - クラス :: のインスタンスにリクエストを渡す *QgsServerInterface*
  - プラグインを呼び出す：FUNC： ‘requestReady‘フィルター
  - **応答がない場合**
    - **サービスは、WMS / WFS / WCS であれば**
      - **WMS / WFS / WCS サーバを作成**

        FUNC：サーバーの呼び出し *executeRequest*‘**と** *possibily* **呼び出す：***FUNC***：** sendResponse‘プラグインフィルタを出力をストリーミングするとき、または要求ハンドラのバイトストリーム出力およびコンテンツタイプを保存します
    - プラグインを呼び出す：FUNC： ‘responseComplete‘フィルター
  - ：func： ‘sendResponse‘フィルタプラグインを呼び出します
  - 要求ハンドラの出力応答

次の段落では、詳細で利用可能なコールバックを記述する。

21.1.1 requestReady

要求の準備ができたときに呼び出されます。受信 URL とデータが解析され、コアサービス（WMS、WFS など）スイッチに入る前に、これはあなたが入力を操作するなどのアクションを実行することができますポイントです。

- 認証/認可
- リダイレクト
- 追加/特定のパラメータ（例えば、型名）を除去
- 例外を発生させます

あなたも** SERVICE **パラメータを変更するので、完全にコアサービスをバイパスすることによって、完全にコアサービスを置き換えることができ（とはいえ、これはあまり意味がないということ）。

21.1.2 sendResponse

出力が送られるたびに呼び出され** FCGI ** *stdout*‘**コアサービスは、そのプロセスを終了し** *responseComplete* **フックが呼ばれた後、しかし、いくつかの例を** *XML* **にした後、これが正常に行われている（そしてそこから、クライアントへ）ストリーミング** *XML* **の実装が必要とされたことをとても巨大になることができ、この場合、（***WFS GetFeature* **のは、そのうちの一つである）：***func* **が：** *‘sendResponse*‘**は応答が完了する前に複数回呼び出さ（と前にされた：***FUNC***：** responseComplete‘が呼び出されます）。FUNC：明白な結果はつまり *sendResponse*‘**は正常に一度呼び出されますが、非常に複数回呼び出されるかもしれないし、その場合には（そして唯一のそのような場合には）それはまたの前に呼び出されます：***FUNC***：** responseComplete‘。

：func が： *sendResponse*‘**は、コアサービスの出力を直接操作のために、しばらく最高の場所です：***FUNC***：** responseComplete‘は、また、一般的なオプションです：FUNC： ‘sendResponse‘は、ストリーミングサービスの場合の唯一の実行可能な選択肢です。

21.1.3 responseComplete

（ヒットした場合）コアサービスは、そのプロセスを終了するときに一度だけ呼び出され、要求がクライアントに送信する準備ができています。FUNC：FUNC :: *sendResponse*‘**呼ばれたかもしれないストリーミングサービス（または他のプラグインフィルター）を除いて上述したように、これは通常の前に呼び出された** sendResponse‘を早く。

：FUNC：‘responseComplete‘は、新しいサービスの実装（WPS またはカスタムサービス）を提供し、コアサービスからの出力を直接操作を実行するために理想的な場所である（例えば WMS の画像に透かしを追加するため)。

21.2 プラグインから例外を上げます

いくつかの作品は、まだこのトピックに行う必要があります：現在の実装では、設定によって処理し、未処理の例外を区別することができます：class： *QgsRequestHandler*‘**プロパティのインスタンスへ：クラス：** QgsMapServiceException‘、このようメイン C ++のコードをキャッチすることができます Python の例外を取り扱い、未処理の例外を無視（またはそれ以上を：それらをログに記録)。

このアプローチは、基本的に動作しますが、それは非常に「神託」ではありません：より良いアプローチは、Python コードから例外が発生し、それらが処理されるための C ++ループに湧き上がっ見ることであろう。

21.3 サーバー・プラグインを書く

参考文献： *developing_plugins*、単に追加の（または代替）インタフェースを提供する：に記載されているようにちょうど標準 QGIS Python のプラグインであるプラグインサーバクラス：‘ QgisInterface‘インスタンス典型的 QGIS デスクトッププラグインを介して QGIS アプリケーションへのアクセスを有していますクラス :: *QgsServerInterface*、サーバプラグインもへのアクセスをしています。

プラグインは、サーバ・インタフェースを持つ QGIS サーバーを指示するには、特別なメタデータエントリは（metadata.txt‘ ‘に）必要とされている:

```
server=True
```

ここでは（より多くの例フィルタ付き）議論プラグインを例は github の上で提供されています：‘QGIS の HelloServer 例プラグイン<https://github.com/elpaso/qgis-helloserver> ‘ _

21.3.1 プラグインファイル

ここに私たちの例のサーバー・プラグインのディレクトリ構造です

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

21.3.2 __init__.py

このファイルは、Python のインポートシステムで必要とされます。：func：‘serverClassFactory サーバーの起動時にプラグインが QGIS Server にロードされるときに呼び出される（）‘関数、また、QGIS Server は、このファイルが含まれていることが必要です。クラス :: ‘QgsServerInterface‘とプラグインのクラスのインスタンスを返す必要がありますそれはのインスタンスへの参照を受け取ります。これは例では、‘__init __ プラグインをする方法です。py‘がどのように見える:

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

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

21.3.3 HelloServer.py

魔法が起こると、これは魔法がどのように見えるかであるところである：(例：ファイル： *HelloServer.py*）

サーバー・プラグインは通常、QgsServerFilter と呼ばれるオブジェクトに詰め込ま一回の以上のコールバックで構成されています。

各：クラス： ‘QgsServerFilter‘は、次のコールバックの一つ以上を実装します。

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

次の例では、**サービスが**パラメータが等しい場合には、HelloServer *！*印刷し、最小限のフィルタを実装する「HELLO」：

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

class HelloFilter(QgsServerFilter):

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

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

フィルタは、次の例のように** ** serverIface に登録する必要があります:

```
class HelloServerServer:
    def __init__(self, serverIface):
        # Save reference to the QGIS server interface
        self.serverIface = serverIface
        serverIface.registerFilter( HelloFilter, 100 )
```

FUNC：の第 2 のパラメータは ‘registerFilter‘は、同じ名前（低い優先順位が最初に呼び出された）とのコールバックの順序を定義する優先順位を設定することを可能にします。

3 つのコールバックを使用することにより、プラグインは、入力および/またはさまざまな方法でサーバーの出力を操作することができます。クラス： *QgsServerInterface*、：クラス：‘ QgsRequestHandler‘がコアに入る前に、入力パラメータを変更するために使用できる方法の多くを有する貫通 *QgsRequestHandler*：クラス：すべての瞬間に、プラグインのインスタンスは、へのアクセスを有しますサーバの処理（：FUNC：使用して *requestReady*）または要求がコアサービスによって処理された後（使用して：FUNC：‘ sendResponse‘）。

次の例は、いくつかの一般的なユースケースをカバーします：

21.3.4 入力を変更します

例えば、プラグインは、新しいパラメータが中に注入され、この例では、クエリ文字列からの入力パラメータを変更する試験例を含んでいる（既に解析された）'parameterMap'は、このパラメータはで、コアサービス（WMS など）によって、次に表示されていますコアサービス処理の終了は、パラメータがまだあることを確認してください:

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

class ParamsFilter(QgsServerFilter):

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

    def requestReady(self):
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        request.setParameter('TEST_NEW_PARAM', 'ParamsFilter')

    def responseComplete(self):
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        if params.get('TEST_NEW_PARAM') == 'ParamsFilter':
            QgsMessageLog.logMessage("SUCCESS - ParamsFilter.responseComplete", 'plugin', QgsMess
        else:
            QgsMessageLog.logMessage("FAIL    - ParamsFilter.responseComplete", 'plugin', QgsMess
```

これは、ログファイルに見るものの抽出物である:

```
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloServerServe
src/core/qgsmessagelog.cpp: 45: (logMessage) [1ms] 2014-12-12T12:39:29 Server[0] Server plugin He
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 Server[0] Server python pl
src/mapserver/qgsgetrequesthandler.cpp: 35: (parseInput) [0ms] query string is: SERVICE=HELLO&requ
src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [1ms] inserting pair S
src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [0ms] inserting pair R
src/mapserver/qgsserverfilter.cpp: 42: (requestReady) [0ms] QgsServerFilter plugin default request
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.reque
src/mapserver/qgis_map_serv.cpp: 235: (configPath) [0ms] Using default configuration file path: /
src/mapserver/qgshttprequesthandler.cpp: 49: (setHttpResponse) [0ms] Checking byte array is ok to
src/mapserver/qgshttprequesthandler.cpp: 59: (setHttpResponse) [0ms] Byte array looks good, setti
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.resp
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] SUCCESS - Params
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] RemoteConsoleFil
src/mapserver/qgshttprequesthandler.cpp: 158: (sendResponse) [0ms] Sending HTTP response
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.send
```

13 行目の「SUCCESS」の文字列は、プラグインがテストに合格したことを示しています。

同じ手法ではなく、コア 1 のカスタムサービスを利用するために利用することができます：あなたは、たとえば** WFS をスキップすることができだけに** SERVICE **パラメータを変更することで** ** ** SERVICE 要求または任意の他のコア要求別の何かとコアサービスはスキップされます、あなたは（これはここでは、以下に説明される）出力にカスタム結果を注入し、クライアントに送信できます。

21.3.5 出力の変更または交換

透かしフィルタの例は、WMS コアサービスによって生成された WMS 画像の上に透かし画像を加算した新たな画像と WMS 出力を交換する方法を示し:

```
import os

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


class WatermarkFilter(QgsServerFilter):

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

    def responseComplete(self):
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        # Do some checks
        if (request.parameter('SERVICE').upper() == 'WMS' \
                and request.parameter('REQUEST').upper() == 'GETMAP' \
                and not request.exceptionRaised() ):
            QgsMessageLog.logMessage("WatermarkFilter.responseComplete: image ready %s" % request
            # Get the image
            img = QImage()
            img.loadFromData(request.body())
            # Adds the watermark
            watermark = QImage(os.path.join(os.path.dirname(__file__), 'media/watermark.png'))
            p = QPainter(img)
            p.drawImage(QRect( 20, 20, 40, 40), watermark)
            p.end()
            ba = QByteArray()
            buffer = QBuffer(ba)
            buffer.open(QIODevice.WriteOnly)
            img.save(buffer, "PNG")
            # Set the body
            request.clearBody()
            request.appendBody(ba)
```

この例では**パラメータ値がチェックされ** SERVICE 着信要求が** WMS ** GetMap リクエスト**と例外がこの中に（WMS 以前に実行プラグインによってまたはコアサービスによって設定されていないである場合ケース)、WMS は、生成された画像は、出力バッファから取得され、透かし画像が追加されます。最後のステップは、出力バッファをクリアして、新たに生成された画像に置き換えることです。実世界の状況で、我々はまた、代わりにどのような場合に PNG を返す要求された画像の種類を確認する必要があることに注意してください。

21.4 アクセス制御プラグイン

21.4.1 プラグインファイル

ここに私たちの例のサーバー・プラグインのディレクトリ構造です:

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

21.4.2 __init__.py

このファイルは、Python のインポートシステムで必要とされます。FUNC：サーバーの起動時にプラグインが QGIS Server にロードされるときに呼び出される'serverClassFactory（）'関数、すべての QGIS サーバプラグインのように、このファイルが含まれています。クラス :: 'QgsServerInterface'とプラグインのクラスのインスタンスを返す必要がありますそれはのインスタンスへの参照を受け取ります。これは例では、'__init __プラグインの方法です py'が見えます。: 67

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

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

21.4.3 AccessControl.py

```
class AccessControl(QgsAccessControlFilter):

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

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

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

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

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

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

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

この例では、皆のための完全なアクセスを提供します。

これは、ログオンしているかを知るためのプラグインの役割です。

これらすべての方法で私達は、層ごとの制限をカスタマイズできるようにするには、引数の層を持っています。

21.4.4 層のフィルタ式

結果を制限するために式を追加するために使用し、例えば：

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

属性の役割がある機能に制限するには、「ユーザー」に等しいです。

21.4.5 layerFilterSubsetString

以前よりも同じですが、(データベース内で実行) `` SubsetString``を使用

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

属性の役割がある機能に制限するには、「ユーザー」に等しいです。

21.4.6 layerPermissions

層へのアクセスを制限します。

性質を持っているタイプ `` QgsAccessControlFilter.LayerPermissions``のオブジェクトを返します：

- `` canRead``は `` GetCapabilities``で彼を見るために、アクセスを読みました。
- `` canInsert``は、新しい機能を挿入することができるようにします。
- `` canUpdate``は、機能を更新することができるようにします。
- `` candelete``は、機能を削除することができるようにします。

Example:

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

読み取り専用のアクセスのすべてを制限します。

21.4.7 authorizedLayerAttributes

属性の特定のサブセットの可視性を制限するために使用します。

引数の属性が表示属性の現在のセットを返します。

Example:

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

「役割」属性を非表示にします。

21.4.8 allowToEdit

これは、機能のサブセットに編集を制限するために使用されます。

これは、 `` WFS-Transaction``プロトコルで使用されています。

Example:

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

値のユーザーを持つ属性の役割を持っているだけの機能を編集することができます。

21.4.9 cacheKey

QGIS サーバは、あなたがこの方法で役割を返すことができる役割ごとにキャッシュを持っている能力のキャッシュを維持します。またはリターン ‘‘ None‘‘には、完全にキャッシュを無効にします。