PyQGIS 3.40 developer cookbook

QGIS Project

2025 年 04 月 03 日

目 次

第1章　はじめに

この文書は、チュートリアルとリファレンスガイドの両方を意図して書かれています。可能な事例をすべて網羅しているわけではありませんが、主要な機能を概観するには役に立つはずです。

Python のサポートは QGIS 0.9 で初めて導入されました。デスクトップ版 QGIS で Python を利用するのにはいくつかの方法があります（この後のセクションで説明します)。

- QGIS に付属する Pyton コンソールでのコマンドの実行
- プラグインの作成と利用
- QGIS 起動時の Python コードの自動実行
- プロセシングアルゴリズムの作成
- QGIS の「式」で使用する関数の作成
- QGIS API を利用するカスタムアプリケーションの作成

QGIS サーバでも Python プラグインを含む Python バインディングが提供されています (参照 *QGIS Server と Python*) そして Python バインディングを使うことによって Python アプリケーションに QGS サーバを埋め込むこともできます。

QGIS ライブラリのクラスのドキュメントである 完全な QGIS C++ API リファレンスがあります。 Python 用の QGIS API（pyqgis） は C ++ API とほぼ同じです。

頻出タスクの実行方法を学習するには、 プラグインのリポジトリ から既存のプラグインをダウンロードして、そのコードを研究するのもよい方法です。

1.1 Python コンソールでのスクリプティング

QGIS はスクリプティングのための Python console を内蔵しています。**プラグイン** → *Python* **コンソール** メニューから開くことができます。

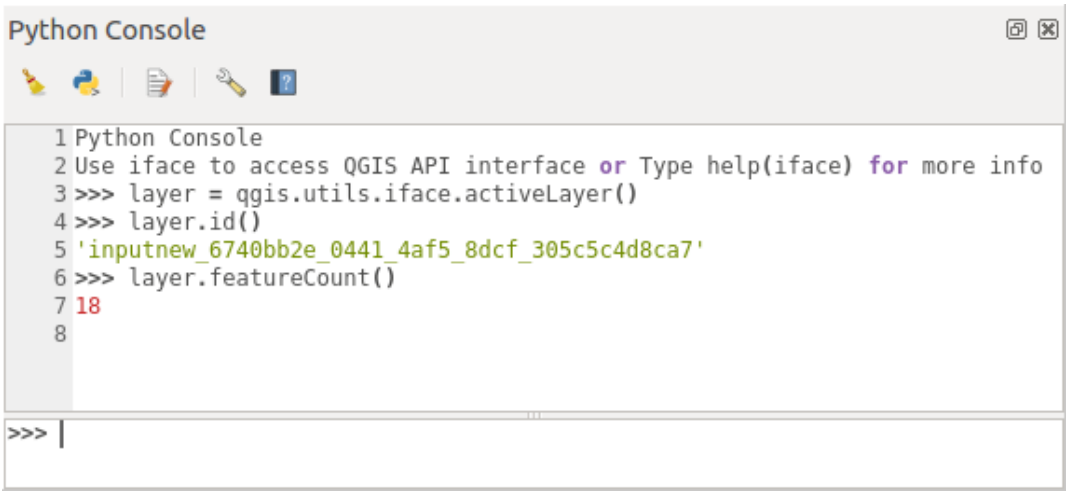


図 1.1: QGIS Python コンソール

上のスクリーンショットは、レイヤリストで現在選択されているレイヤを取得し、その ID を表示し、オプションでベクタレイヤの場合は地物数を表示する方法を示しています。QGIS 環境とのやりとりのために、`QgisInterface` のインスタンスである `iface` 変数があります。このインタフェースにより、地図 キャンバス、メニュー、ツールバー、および QGIS アプリケーションのその他の部分へのアクセスが可能になります。

ユーザーの便宜のために、コンソールの起動時に次のステートメントが実行されます（将来的にはさらに初期コマンドを設定できるようになります）

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

コンソールを頻繁に使用する場合は、コンソールを起動するためのショートカットを設定すると便利な場合があります（ **設定** → **キーボードショートカット**... 内）

1.2 Python プラグイン

QGIS の機能はプラグインを使って拡張することができます。プラグインは Python で書くことができます。C++プラグインに比較しての主要な利点は配布の単純さ（プラットフォームごとにコンパイルする必要がありません）と開発の容易さです。

様々な機能をカバーする多くのプラグインが 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。プラグインとプラグイン開発の詳細については、 Python プラグイン ページを参照してください。

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

注釈: Python プラグインは QGIS サーバーでも使うことができます。より詳しくは *QGIS Server と Python* をご覧ください。

1.2.1 プロセシングプラグイン

プロセシングプラグインはデータを処理するために使うことができます。Python プラグインよりも開発が簡単で、より具体的で軽量です。 **プロセシングプラグインを書く** では、プロセシングアルゴリズムを使用することが適切な場合や、どのように開発すればよいかを説明します。

1.3 QGIS 起動時に Python コードを実行する

QGIS が起動する度に Python コードを実行する方法は複数あります。

1. startup.py スクリプトを作る
2. 既存の Python ファイルを `PYQGIS_STARTUP` 環境変数に設定する
3. `--code init_qgis.py` パラメータを使って起動スクリプトを指定する。

1.3.1 `startup.py` ファイル

QGIS が起動する度、ユーザの Python ホームディレクトリとシステムパスのリストに :file:`startup.py` を探します。そのファイルがあれば、埋め込み Python インタプリタを使って実行されます。

通常、ユーザーのホームディレクトリにあるパスは、以下のようになります:

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

デフォルトのシステムパスは、オペレーティングシステムに依存します。自分に合ったパスを見つけるには、Python コンソールを開いて `QStandardPaths.standardLocations(QStandardPaths.AppDataLocation)` を実行すると、デフォルトディレクトリのリストが表示されます。

`startup.py` スクリプトは、QGIS で python を初期化した直後、アプリケーション起動の初期に実行されます。

1.3.2 PYQGIS_STARTUP 環境変数

既存の Python ファイルのパスを PYQGIS_STARTUP 環境変数に設定すると、QGIS の初期化が完了する直前に Python コードを実行することができます。

このコードは、QGIS の初期化が完了する前に実行されます。この方法は、望ましくないパスが含まれている可能性のある sys.path をクリーンアップする場合や、仮想環境を必要としないで初期環境を分離/ロードする場合に非常に役立ちます。 例 Mac の homebrew または MacPorts のインストール。

1.3.3 --code パラメータ

QGIS の起動パラメータとして実行するカスタムコードを提供することができます。そのためには、例えば `qgis_init.py` のような Python ファイルを作成し、`qgis --code qgis_init.py` を用いてコマンドラインから QGIS を実行・起動するようにします。

`--code` で提供されるコードは、アプリケーションのコンポーネントがロードされた後、QGIS の初期化フェーズの後半に実行されます。

1.3.4 Python の追加引数

スクリプト `--code` や実行される他の Python コードに追加の引数を与えるには、`--py-args` 引数を使用することができます。`--py-args` の後で（もしあれば） `--` arg の前に来る引数は、Python に渡されますが、QGIS アプリケーション自体では無視されます。

以下の例では、`myfile.tif` は Python の `sys.argv` で利用できますが、QGIS では読み込まれません。一方、`otherfile.tif` は QGIS によってロードされますが、`sys.argv` には存在しません。

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

Python から全てのコマンドラインパラメータにアクセスしたい場合は、`QCoreApplication.arguments()` を使用することができます。

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

1.4 Python アプリケーション

多くの場合、プロセスを自動化するためのスクリプトを作成すると便利です。 PyQGIS を使用すると、これは完全に可能です--- qgis.core モジュールをインポートし、初期化すると、処理の準備が整います。

または、GIS 機能を使用するインタラクティブなアプリケーションを作成することもできます---測定を実行し、地図を PDF としてエクスポートします...。 qgis.gui モジュールではさまざまな GUI コンポーネントを提供します。特に地図キャンバスウィジェットは、ズーム、パン、その他のカスタムの地図ツールをサポートしてアプリケーションに組み込むことができます。

PyQGIS カスタムアプリケーションやスタンドアロンスクリプトは、ベクタレイヤとラスタレイヤを読み取るための投影情報やプロバイダーなどの QGIS リソースを見つけるように構成する必要があります。 QGIS

リソースは、アプリケーションまたはスクリプトの先頭に数行を追加することで初期化されます。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は似ています。それぞれの例を以下に示します。

注釈: スクリプトの名前に qgis.py を使用しないでください。スクリプトの名前が隠すので Python はバインディングをインポートできなくなります。

1.4.1 スタンドアロンスクリプトで PyQGIS を使用する

スタンドアロンスクリプトを始めるには、スクリプトの初めに QGIS 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.

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

まず、 qgis.core モジュールをインポートし、プレフィックスパスを設定します。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。 `setPrefixPath()` メソッドを呼び出すことでスクリプトで構成されます。 `setPrefixPath()` の 2 番目の引数は `True` に設定されます。これはデフォルトのパスが使用されることを指定しています。

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。お手元のシステムに対するインストールパスを見つける最も簡単な方法は、QGIS 内から *Python* **コンソールでのスクリプティング** を使用し、実行からの出力を見ることです：

```
QgsApplication.prefixPath()
```

プレフィックスパスを設定した後、 `QgsApplication` への参照を変数 `qgs` に保存します。 2 番目の引数は `False` に設定され、スタンドアロンスクリプトを記述しているため、GUI を使用する予定がないことを指定します。 `QgsApplication` を設定した状態で、 `initQgis()` メソッドを呼び出して QGIS データプロバイダとレイヤーレジストリをロードします。

```
qgs.initQgis()
```

QGIS が初期化されたら、残りのスクリプトを書く準備ができています。最後に、 `exitQgis()` を呼び出して、データプロバイダとレイヤーレジストリをメモリから削除して終了します。

```
qgs.exitQgis()
```

1.4.2 カスタムアプリケーションで PyQGIS を使用する

スタンドアロンスクリプトで *PyQGIS* を使用する とカスタム PyQGIS アプリケーションが違うのは、`QgsApplication <qgis.core.QgsApplication>` ` をインスタンス化するときの 2 番目の引数だけです。 :const:`False の代わりに True を渡して、GUI を使用する予定であることを示します。

```
from qgis.core import *

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

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

qgs = QgsApplication([], True)

# load providers
qgs.initQgis()

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

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

これで QGIS API を使用して、レイヤをロードして処理を行ったり、マップキャンバスを使用して GUI を起動したりすることができるようになりました。可能性は無限大です:-)

1.4.3 カスタムアプリケーションを実行する

QGIS ライブラリと適切な Python モジュールがよく知られた場所にない場合、どこに検索をかけるかをシステムに指示する必要があります:

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

これは、環境変数 PYTHONPATH を設定することで修正することができます。以下のコマンドでは、<qgispath> をあなたの実際の QGIS のインストールパスに置き換えてください:

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

これで、PyQGIS モジュールへのパスがわかりましたが、それらは qgis_core および qgis_gui ライブラリに依存しています（Python モジュールはラッパーとしてのみ機能します）。これらのライブラリへのパスがオペレーティングシステムに認識されていない可能性があり、その後、インポートエラーが再度発生します（メッセージはシステムによって異なる場合があります）:

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

ダイナミックリンカーの検索パスに QGIS ライブラリが存在するディレクトリを追加することで修正します:

- Linux では: **export LD_LIBRARY_PATH=/<qgispath>/lib**
- Windows では: **set PATH=C:\<qgispath>\bin;C:\<qgispath>\apps\<qgisrelease>\bin;%PATH%** <qgisrelease> はターゲットにしているリリースのタイプで置き換えてください（例 qgis-ltr, qgis, qgis-dev）

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:

- アプリケーションをインストールする前に、ユーザーに QGIS をインストールするように要求します。アプリケーションインストーラーは QGIS ライブラリのデフォルトの場所を探し、見つからない場合はユーザーがパスを設定できるようにする必要があります。このアプローチには、より単純であるという利点がありますが、ユーザーはより多くの手順を実行する必要があります。
- 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、ユーザーが追加ソフトウェアをダウンロードしてインストールするという負荷を避けられるでしょう。

2 つの展開モデルを混在させることができます。 Windows と macOS でスタンドアロンアプリケーションを提供できますが、Linux の場合、GIS のインストールはユーザーとそのパッケージマネージャに任せます。

1.5 PYQt と SIP の技術メモ

Python が選ばれたのは、それがスクリプト作成に最も好まれる言語の 1 つであるためです。 QGIS 3 の PyQGIS バインディングは、SIP と PyQt5 に依存しています。より広く使用されている SWIG の代わりに SIP を使用する理由は、QGIS コードが Qt ライブラリに依存しているためです。 Qt (PyQt) の Python バインディングは SIP を使用して行われ、これにより PyQGIS と PyQt のシームレスな統合が可能になります。

第2章　プロジェクトをロードする

ヒント: pyqgis コンソールを使わない場合、このページにあるコードスニペットは次のインポートが必要です:

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

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

プラグインや、スタンドアローンの QGIS Python アプリケーションから、既存のプロジェクトをロードする必要のあることがあります（参照： *Python* **アプリケーション** ）。

プロジェクトを現在の QGIS アプリケーションにロードするには、 `QgsProject` クラスのインスタンスを作成する必要があります。これはシングルトンクラスなので、それを行うには `instance()` メソッドを使わなければなりません。 `read()` メソッドを呼び出して、読み込むプロジェクトのパスを渡すことができます。

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

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

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

プロジェクトに変更（たとえばレイヤの追加や削除）を加え、その変更を保存する必要がある場合は、プロ

ジェクトインスタンスの write() メソッドを呼び出します。 write() メソッドにパスを指定すれば、プロジェクトを新しい場所に保存することもできます。

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

read() と write() の両方の関数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。

注釈: 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、ロードされたプロジェクトをキャンバスと同期させるために、 QgsLayerTreeMapCanvasBridge を以下の例のようにインスタンス化する必要があります:

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

2.1 正しくないパスを解決する

プロジェクトにロードされたレイヤが別の場所に移動されることがあります。プロジェクトが再びロードされたとき、全てのレイヤのパスが壊れてしまいます。QgsPathResolver クラスは、プロジェクト内のレイヤパスを書き換える手助けをします。

setPathPreprocessor() メソッドは、ファイル参照やレイヤソースのパスやデータソースを解決する前にそれを操作する、カスタムパスプリプロセッサ関数を設定することを可能にします。

プロセッサ関数は、1 つの文字列引数（元のファイルパスまたはデータソースを表す）を受け取り、このパスの処理済みバージョンを返します。パスプリプロセッサ関数は、不正レイヤハンドラの **前に** 呼ばれます。複数のプリプロセッサが設定されている場合、それらは最初に設定された順序に基づいて順番に呼び出されます。

使用例：

1. 旧パスを置き換え：

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

   QgsPathResolver.setPathPreprocessor(my_processor)
   ```

2. データベースのホストアドレスを新しいもので置き換え：

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

QgsPathResolver.setPathPreprocessor(my_processor)
```

3. 保存されているデータベースのクレデンシャルを新しいものに置き換え：

```
def my_processor(path):
    path= path.replace("user='gis_team'", "user='team_awesome'")
    path = path.replace("password='cats'", "password='g7as!m*'")
    return path

QgsPathResolver.setPathPreprocessor(my_processor)
```

同様に、パスライタ機能として `setPathWriter()` メソッドが利用可能です。

パスを変数で置き換える例:

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

QgsPathResolver.setPathWriter(my_processor)
```

どちらのメソッドも id を返し、追加されたプリプロセッサやライタを削除するために使うことができます。`removePathPreprocessor()` と `removePathWriter()` を参照してください。

2.2 フラグで速度を向上させる

完全に機能するプロジェクトを使う必要はなく、特定の理由でのみアクセスしたい場合、フラグを使うと便利な場合があります。フラグの完全なリストは `ProjectReadFlag` の下にあります。複数のフラグを一緒に追加することができます。

例として、実際のレイヤやデータを気にせず、単にプロジェクトにアクセスしたい場合（例 レイアウトや 3D ビューの設定など）、`DontResolveLayers` フラグを使ってデータ検証ステップをバイパスし、不良レイヤダイアログが表示されないようにすることができます。次のようにすることができます：

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

さらにフラグを追加するには、Python の Bitwise OR 演算子 (|) を使用する必要があります。

第3章　レイヤをロードする

ヒント: このページのコードスニペットは、以下のインポートが必要です:

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

データのレイヤを開きましょう。QGIS はベクタおよびラスタレイヤを認識できます。加えてカスタムレイヤタイプを利用することもできますが、それについてここでは述べません.

3.1 ベクタレイヤ

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

geopackage ベクタレイヤなら次のようになります:

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

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

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

これは、新しいレイヤを作成し、それを現在の QGIS プロジェクトに追加します（レイヤリストに表示されます）。この関数は、レイヤのインスタンスを返すか、レイヤを読み込むことができなかった場合は None を返します。

以下のリストはベクタデータプロバイダー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:

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

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

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

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

  - dxf（データソース uri 中の内部オプションに注意）の場合:

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

- PostGIS 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含む文字列です。

  QgsDataSourceUri クラスを使用すると、この文字列を生成することができます。QGIS が Postgres サポートとコンパイルされていない場合、このプロバイダを利用できないことに注意してください:

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

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

注釈: uri.uri(False) に渡される False 引数は、認証構成パラメーターの拡張を防ぎます。もし何も認証構成を使用していなければ、この引数は何の違いもありません。

- CSV などの区切りテキストファイル --- セミコロンを区切り文字として、X 座標をフィールド「x」、Y 座標をフィールド「y」としたファイルを開くには、以下のようにします:

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

注釈: プロバイダーの文字列は URL として構造化されているので、パスには `file://` という接頭辞を付ける必要があります。また、 `x` や `y` フィールドの代わりに WKT（well-known text）形式のジオメトリを使用でき、座標参照系を指定できます。例えば:

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

- GPX ファイル---「GPX」データプロバイダーは、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）を URL の一部として指定する必要があります：

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

- SpatiaLite データベース --- PostGIS データベースと同様に、データソースの識別子を生成するために `QgsDataSourceUri` を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')
QgsProject.instance().addMapLayer(vlayer)
```

- MySQL WKB ベースのジオメトリ、GDAL 経由 --- データソースは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" )
QgsProject.instance().addMapLayer(vlayer)
```

- WFS 接続: 接続は URI で定義され、 `WFS` プロバイダを使用します:

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

URI は標準の urllib ライブラリを使用して作成できます:

```
import urllib

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

注釈: 以下の例のように、 QgsVectorLayer インスタンスに対して setDataSource() を呼び出して、既存のレイヤのデータソースを変更することができます:

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

del(vlayer)
```

3.2 ラスタレイヤ

ラスタファイルのアクセスには、GDAL ライブラリを使用します。GDAL は様々なファイル形式をサポートしています。一部のファイルを開く際に問題が発生する場合は、使っている GDAL が特定のフォーマットをサポートしているかどうかを確認してください（デフォルトですべてのフォーマットが利用可能なわけではありません）。ファイルからラスタを読み込むには、そのファイル名と表示名を指定します:

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

geopackage からラスタを読み込むには:

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

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

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

ベクタレイヤと同様に、ラスタレイヤは QgisInterface オブジェクトの addRasterLayer 関数を使って読み込むことができます:

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

これは、新しいレイヤを作成し、現在のプロジェクトに追加する（レイヤリストに表示させる）作業を 1 ステップで行います。

PostGIS ラスタを読み込むには:

PostGIS ラスタは、PostGIS ベクタと同様に、URI 文字列を使用してプロジェクトに追加することができます。データベース接続パラメータ用の文字列の再利用可能な辞書を保持することは効率的です。これにより、該当する接続の辞書を簡単に編集することができます。この辞書は、'postgresraster' プロバイダメタデータオブジェクトを使用して URI にエンコードされます。その後、ラスタをプロジェクトに追加することができます。

```
uri_config = {
    # database parameters
    'dbname':'gis_db',      # The PostgreSQL database to connect to.
    'host':'localhost',     # The host IP address or localhost.
    'port':'5432',          # The port to connect on.
    'sslmode':QgsDataSourceUri.SslDisable, # SslAllow, SslPrefer, SslRequire,␣
↪SslVerifyCa, SslVerifyFull
    # user and password are not needed if stored in the authcfg or service
    'authcfg':'QconfigId',  # The QGIS athentication database ID holding connection␣
↪details.
    'service': None,         # The PostgreSQL service to be used for connection to␣
↪the database.
    'username':None,        # The PostgreSQL user name.
    'password':None,        # The PostgreSQL password for the user.
    # table and raster column details
    'schema':'public',      # The database schema that the table is located in.
    'table':'my_rasters',   # The database table to be loaded.
    'geometrycolumn':'rast',# raster column in PostGIS table
    'sql':None,             # An SQL WHERE clause. It should be placed at the end of␣
↪the string.
```

(次のページに続く)

(前のページからの続き)

```
    'key':None,             # A key column from the table.
    'srid':None,            # A string designating the SRID of the coordinate reference system.
    'estimatedmetadata':'False', # A boolean value telling if the metadata is estimated.
    'type':None,            # A WKT string designating the WKB Type.
    'selectatid':None,      # Set to True to disable selection by feature ID.
    'options':None,         # other PostgreSQL connection options not in this list.
    'enableTime': None,
    'temporalDefaultTime': None,
    'temporalFieldIndex': None,
    'mode':'2',             # GDAL 'mode' parameter, 2 unions raster tiles, 1 adds tiles separately (may require user input)
}
# remove any NULL parameters
uri_config = {key:val for key, val in uri_config.items() if val is not None}
# get the metadata for the raster provider and configure the URI
md = QgsProviderRegistry.instance().providerMetadata('postgresraster')
uri = QgsDataSourceUri(md.encodeUri(uri_config))

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

ラスタレイヤも WCS サービスから作成できます:

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

以下は、WCS URI が含むことのできるパラメータの説明です:

WCS URI は & で区切られた **key=value** のペアで構成されています。URL のクエリ文字列と同じ形式で、同じようにエンコードされます。 `QgsDataSourceUri` は、特殊文字が適切にエンコードされるように URI を組み立てるために使用されるべきものです。

- **url** （必須）: WCS サーバーの URL。WCS の各バージョンでは、**GetCapabilities** のバージョンに異なるパラメータ名を使用しているため、URL には VERSION を使用しないでください（param version を参照）。
- **identifier** （必須）: Coverage name
- **time** （オプション）: 時間位置または時間帯 (beginPosition/endPosition[/timeResolution])
- **format** （オプション）: サポートされているフォーマット名。デフォルトは名前に tif を伴う最初のサポートされているフォーマット、または最初のサポートされているフォーマット。
- **crs** （オプション）: AUTHORITY:ID 形式による CRS 例. EPSG:4326。デフォルトは EPSG:4326（サポートされている場合）または最初のサポートされている CRS。

- **username** （オプション）：基本認証の Username。
- **password** （オプション）：基本認証のパスワード。
- **IgnoreGetMapUrl** （オプション、ハック）：指定されたとき（1 を参照）、GetCapabilities が布告した GetCoverage URL を無視する。サーバーが正しく構成されたいないときに必要かもしれない。
- **InvertAxisOrientation** （オプション、ハック）：指定されたとき（1 に設定）、GetCoverage 要求で軸を切り替えます。サーバーが間違った軸の順序を使用している場合、地理的な CRS のために必要な場合があります。
- **IgnoreAxisOrientation**（オプション、ハック）：指定された場合（1 に設定）、地理的 CRS の WCS 標準に従って軸の向きを反転させないようにする。
- **cache** （オプション）：QNetworkRequest::CacheLoadControl で説明されているように、キャッシュの読み込みを制御します。ただし、AlwaysCache で失敗した場合は PreferCache としてリクエストを再送します。可能な値： AlwaysCache、PreferCache、PreferNetwork、AlwaysNetwork。デフォルトは AlwaysCache です。

別の方法としては、WMS サーバーからラスターレイヤー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---どのレイヤが必要か知っている必要があります。

```
urlWithParams = "crs=EPSG:4326&format=image/png&layers=continents&styles&url=https://
↪demo.mapserver.org/cgi-bin/wms"
rlayer = QgsRasterLayer(urlWithParams, 'some layer name', 'wms')
if not rlayer.isValid():
  print("Layer failed to load!")
```

3.3 QgsProject インスタンス

開いたレイヤをレンダリングに使用したい場合は、 `QgsProject` インスタンスに追加することを忘れないようにしてください。`QgsProject` インスタンスはレイヤの所有権を持ち、後でアプリケーションのどの部分からでも一意の ID によってアクセスすることができます。レイヤがプロジェクトから削除されると、レイヤも削除されます。レイヤは QGIS インターフェイスでユーザーが削除することもできますし、Python で `removeMapLayer()` メソッドを使用しても削除できます。

現在のプロジェクトにレイヤを追加するには、 `addMapLayer()` メソッドを用います:

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

絶対位置にレイヤーを追加する:

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

レイヤを削除したい場合は、 `removeMapLayer()` メソッドを使用します:

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

上記のコードでは、レイヤの ID を渡していますが（レイヤの `id()` メソッドで取得できます）、レイヤオブジェクトそのものを渡すことも可能です。

ロードされたレイヤのリストとレイヤ ID については、 `mapLayers()` メソッドを使用してください:

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

第4章　目次（TOC）へのアクセス

ヒント: pyqgis コンソールを使わない場合、このページにあるコードスニペットは次のインポートが必要です:

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

さまざまなクラスを使用して、TOC にロードされているすべてのレイヤーにアクセスし、それらを使用して情報を取得できます：

- QgsProject
- QgsLayerTreeGroup

4.1 QgsProject クラス

QgsProject を使うと、TOC とロードされた全てのレイヤに関する情報を取得することができます。

QgsProject の「インスタンス」を作成し、そのメソッドを使用してロードされたレイヤを取得する必要があります。

メインメソッドは mapLayers() です。ロードされたレイヤの辞書を返します:

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

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

辞書の keys はユニークなレイヤ ID であり、 values は関連するオブジェクトです。

これで、レイヤに関するその他の情報を簡単に得ることができるようになりました:

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

(次のページに続く)

(前のページからの続き)

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

print(layers_list)
```

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

また、レイヤの名前で TOC を照会することも可能です:

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

注釈: 一致するすべてのレイヤを含んだリストが返されるため、`[0]` でインデックスを作成して、この名前の最初のレイヤを取得します。

4.2 QgsLayerTreeGroup クラス

レイヤツリーは、ノードで構築された古典的なツリー構造です。現在、ノードには 2 つのタイプがあります：グループノード（`QgsLayerTreeGroup`）とレイヤノード（`QgsLayerTreeLayer`）。

注釈: for more information you can read these blog posts of Martin Dobias: Part 1 Part 2 Part 3

プロジェクトのレイヤツリーには `QgsProject` クラスの `layerTreeRoot()` メソッドで容易にアクセスすることができます:

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

root はグループノードであり、 **子** を持っています：

```
root.children()
```

直接の子のリストが返されます。サブグループの子には、自分の直接の親からアクセスする必要があります。

子の一人を検索することができます:

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

```
<QgsLayerTreeLayer: countries>
```

レイヤは、その（ユニークな） id を使用して取得することもできます:

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

また、グループは名前を使用して検索することもできます：

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

QgsLayerTreeGroup は TOC に関するより多くの情報を得るために使用できる多くの便利なメソッドを備えています:

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

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

では、プロジェクトのレイヤツリーにいくつかのレイヤを追加してみましょう。これには 2 つの方法があります:

1. **明示的な追加** をするには、addLayer() または insertLayer() 関数を使用します:

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

2. **暗黙の追加**：プロジェクトのレイヤツリーはレイヤレジストリに接続されているので、マップレイヤレジストリにレイヤを追加するだけで十分です:

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

QgsVectorLayer と QgsLayerTreeLayer は簡単に切り替えて使うことができます:

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

```
Layer node: <QgsLayerTreeLayer: countries>
Map layer: <QgsVectorLayer: 'countries' (ogr)>
```

グループは addGroup() メソッドで追加できます。以下の例では、前者は TOC の最後にグループを追加し、後者の場合は既存のグループ内に別のグループを追加できます：

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

ノードとグループを移動するには、多くの便利な方法があります。

既存のノードの移動は、次の 3 つのステップで実行されます：

1. 既存ノードをクローニングする
2. クローンしたノードを動かしたい位置に移動する
3. 元のノードを削除する

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

レイヤを凡例の中で動かすのは少し **ややこしい*** です:

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

または既存のグループへ移動させます:

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

(次のページに続く)

(前のページからの続き)

```
# move the cloned layer to the top (0)
group1.insertChildNode(0, myvlclone)
# remove the QgsLayerTreeLayer from its parent
parent.removeChildNode(myvl)
```

グループやレイヤの変更に使えるその他いくつかのメソッド:

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

第5章　ラスタレイヤを使う

ヒント: pyqgis コンソールを使わない場合、このページにあるコードスニペットは次のインポートが必要です:

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

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

5.1 レイヤの詳細

ひとつのラスタレイヤは、1 つまたは複数のラスタバンド（シングルバンドおよびマルチバンドラスタと呼ばれる）で構成されています。1 つのバンドは、値の行列を表します。カラー画像（航空写真など）は、赤、青、緑のバンドで構成されるラスタです。シングルバンドのラスタは、通常、連続変数（標高など）または離散変数（土地利用など）を表します。ラスタレイヤにパレットが付属し、ラスタ値がパレットに格納されている色を参照する場合もあります。

次のコードは `rlayer` が `QgsRasterLayer` オブジェクトであると仮定しています。

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

```
919 619
```

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

```
<QgsRectangle: 20.06856808199999875 -34.27001076999999896, 20.83945284300000012 -33.
↪75077500700000144>
```

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

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

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

```
0
```

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

```
1
```

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

```
Band 1: Height
```

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

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

5.2 レンダラー

ラスタレイヤがロードされると、そのタイプに応じたデフォルトレンダラーが取得されます。これは、レイヤプロパティまたはプログラムによって変更することができます。

現在のレンダラーを問い合わせるには:

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

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

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

```
singlebandgray
```

レンダラーを設定するには、 QgsRasterLayer の setRenderer() メソッドを使います。レンダラークラス（ QgsRasterRenderer から派生したもの）は多数存在します:

- QgsHillshadeRenderer
- QgsMultiBandColorRenderer
- QgsPalettedRasterRenderer
- QgsRasterContourRenderer
- QgsSingleBandColorDataRenderer
- QgsSingleBandGrayRenderer
- QgsSingleBandPseudoColorRenderer

シングルバンドのラスタレイヤは、グレーカラー（低い値＝黒、高い値＝白）、または値に色を割り当てる疑似カラーアルゴリズムで描画することができます。パレットを持つシングルバンドラスタは、そのパレットを使って描画することもできます。マルチバンドレイヤは、通常、バンドを RGB カラーにマッピングして描画します。他には、描画に 1 つのバンドだけを使うこともできます。

5.2.1 シングルバンドラスタ

例えば、緑から黄色までの色（0 から 255 までのピクセル値に対応）を持つシングルバンドのラスタレイヤをレンダリングしたいとします。最初の段階では、 QgsRasterShader オブジェクトを用意し、シェーダ関数を設定します:

```
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)
```

シェーダはカラーマップで指定された色をマッピングします。カラーマップは、ピクセル値と関連する色のリストとして提供されます。補間には 3 つのモードがあります:

- 線形 (Interpolated): ピクセル値の上下のカラーマップエントリから線形補間された色
- 離散 (Discrete): 同じかそれ以上の値を持つ、最も近いカラーマップのエントリから取得された色
- 同値 (Exact): 色は補間されず、カラーマップのエントリと同じ値を持つピクセルのみが描画されます

第 2 ステップでは、このシェーダをラスタレイヤに関連付けます：

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

上のコードにある 1 という数字はバンド番号です（ラスタバンドは 1 から始まるインデックスを持ちます）。

最後に、結果を見るために `triggerRepaint()` メソッドを使わなければなりません：

```
rlayer.triggerRepaint()
```

5.2.2 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、青にマッピングしてカラー画像を作成します（これが `MultiBandColor` という描画スタイルです）。場合によってはこれらの設定を上書きしたいこともあるでしょう。以下のコードは赤のバンド (1) と緑のバンド (2) を入れ替えます：

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

ラスタの視覚化に必要なバンドが 1 つだけの場合は、グレーレベルまたは擬似カラーによる単一バンドの描画を選択できます。

マップを更新して結果を見るには `triggerRepaint()` を使わなければなりません：

```
rlayer_multi.triggerRepaint()
```

5.3 値の検索

ラスタ値は `QgsRasterDataProvider` クラスの `sample()` メソッドを使って問い合わせることができます。`QgsPointXY` と問い合わせたいラスタレイヤのバンド番号を指定する必要があります。メソッドはその値と、結果に応じて `True` または `False` をタプルで返します：

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

ラスタの値を問い合わせるもう一つの方法は、`QgsRasterIdentifyResult` オブジェクトを返す `identify()` メソッドを使うことです。

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

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

```
{1: 323.0}
```

この場合、 results() メソッドは、バンドのインデックスをキー、バンドの値を値とする辞書を返します。例えば、 {1: 323.0} のようになります

5.4 ラスタデータを編集する

ラスタレイヤは QgsRasterBlock クラスを使って作成することができます。例えば、1 ピクセルあたり 1 バイトの 2x2 のラスターブロックを作成するには：

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

ラスタのピクセルは writeBlock() メソッドによって上書きすることができます。0,0 の位置にある既存のラスタデータを 2x2 のブロックで上書きするには：

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

第6章　ベクタレイヤを使う

ヒント: pyqgis コンソールを使わない場合、このページにあるコードスニペットは次のインポートが必要です:

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

from qgis.core.additions.edit import edit

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

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.

ここでのほとんどの作業は QgsVectorLayer クラスのメソッドに基づきます。

6.1 属性に関する情報を取得する

クラス :*QgsVectorLayer <qgis.core.QgsVectorLayer>* オブジェクトに対して fields() を呼び出すことでベクタレイヤに関連するフィールドに関する情報を取得することができます:

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

```
fid Integer64
id Integer64
scalerank Integer64
featurecla String
type String
name String
abbrev String
location String
gps_code String
iata_code String
wikipedia String
natlscale Real
```

displayField() と mapTipTemplate() メソッドは、 maptips タブで使用するフィールドとテンプレートについての情報を提供します。

ベクタレイヤを読み込むと、常にフィールドが Display Name として QGIS によって選択され、HTML Map Tip はデフォルトで空になっています。これらのメソッドを使用すると、簡単に両方を取得することができます:

```
vlayer = QgsVectorLayer("testdata/data/data.gpkg|layername=airports", "Airports layer
↪", "ogr")
print(vlayer.displayField())
```

```
name
```

注釈: 表示名 をフィールドから式に変更する場合、 displayField() の代わりに displayExpression() を使用しなければなりません。

6.2 ベクタレイヤの反復処理

ベクタレイヤの地物を反復することは、最も一般的なタスクの 1 つです。以下は、このタスクを実行するためのシンプルな基本コードの例で、各地物に関するいくつかの情報を表示します。変数 `layer` には `QgsVectorLayer` オブジェクトが含まれていると仮定しています。

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

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

(次のページに続く)

(前のページからの続き)

```
    break
```

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

6.3 地物の選択

QGIS デスクトップでは、地物の選択はさまざまな方法で行うことができます: 地物をクリックする、マップキャンバス上に矩形を描く、または式フィルタを使用する。選択された地物は通常、ユーザーの注意を引くよう、別の色（デフォルトは黄色）でハイライトされます。

プログラムで地物を選択したり、デフォルトの色を変更したりすることが便利な場合もあります。

全ての地物を選択するためには、 `selectAll()` メソッドを使うことができます:

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

式を使用して選択するには、 `selectByExpression()` メソッドを使います:

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

選択色を変更するには、次の例のように、 `QgsMapCanvas` の `setSelectionColor()` メソッドを使うことができます:

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

指定されたレイヤの選択された地物リストに地物を追加するには、 `select()` を呼び出し、地物 ID のリストを渡します:

```
selected_fid = []

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

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

選択を解除するには:

```
layer.removeSelection()
```

6.3.1 属性にアクセスする

属性は名前で参照することができます:

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

```
First feature
```

また、属性はインデックスで参照することもできます。これは名前を使うより少し速いです。例えば、2 番目の属性を取得する場合:

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

```
First feature
```

6.3.2 選択された地物への反復処理

選択された地物のみが必要な場合は、ベクタレイヤの `selectedFeatures()` メソッドを使用することができます:

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

6.3.3 一部の地物の反復処理

もし、レイヤ内のあるエリア内の地物のサブセットを反復処理したい場合、 `getFeatures()` 呼び出しに QgsFeatureRequest> オブジェクトを追加する必要があります。以下はその例です:

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

request = QgsFeatureRequest().setFilterRect(areaOfInterest)

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

高速化のため、多くの場合、地物のバウンディングボックスのみを使用して交差が行われます。しかし、フラグ ExactIntersect を指定することで、交差する地物のみが返されるようにすることができます:

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

;meth:*setLimit() <qgis.core.QgsFeatureRequest.setLimit>* を使用すると、リクエストした地物の数を制限することができます。以下はその例です:

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

```
<qgis._core.QgsFeature object at 0x7f9b78590948>
<qgis._core.QgsFeature object at 0x7faef5881670>
```

上記の例のように空間的なフィルタの代わりに、または加えて、属性ベースのフィルタが必要な場合は、QgsExpression オブジェクトを作って QgsFeatureRequest コンストラクタに渡せばよいでしょう。以下はその例です:

```
# 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)
```

QgsExpression がサポートする構文の詳細については、 **式、フィルタ適用および値の算出** を参照してください。

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 to increase the "speed" of the request
request.setSubsetOfAttributes([0,2])

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

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

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

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

6.4 ベクタレイヤを修正する

大部分のベクタデータプロバイダーは、レイヤデータの編集をサポートしています。プロバイダーによっては、編集操作の一部しかサポートしていないこともあります。どの機能をサポートしているかを知るには、 capabilities() 関数を使ってください。

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

```
The layer supports DeleteFeatures
```

利用可能なすべての機能のリストについては、 API Documentation of QgsVectorDataProvider を参照してください。

capabilitiesString() を使うと、下記の例に見るように、レイヤの機能の説明文をコンマで区切られたリストの形で表示することができます:

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

ベクタレイヤを編集する以下の方法はいずれも、変更が直接、レイヤの裏にあるデータストア（ファイルやデータベースなど）にコミットされます。一時的な変更をしたいだけの場合にどうすればよいかの説明は、次のセクション **ベクタレイヤを編集バッファで修正する** でしているので、以下を飛ばしてそちらに進んでください。

注釈: QGIS の内部（コンソールまたはプラグインのいずれか）で作業している場合、ジオメトリ、スタイル、属性に加えられた変更を確認するために、以下のようにマップキャンバスの強制的な再描画が必要になることもあります:

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

6.4.1 地物の追加

QgsFeature インスタンスをいくつか作成し、プロバイダの QgsVectorDataProvider addFeatures() メソッドにそれらのリストを渡します。このメソッドは、結果 (True または False) 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）という 2 つの値を返します。

地物の属性を設定するには、 QgsFields オブジェクト（ベクタレイヤの fields() メソッドから取得できます）を渡して地物を初期化するか、追加したいフィールド数を渡して initAttributes() を呼び出す方法があります。

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

6.4.2 地物の削除

一部の地物を削除するには、その地物 ID のリストを提供するだけです。

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

6.4.3 地物の修正

地物のジオメトリを変更することも、一部の属性を変更することも可能です。次の例では、まずインデックス 0 と 1 の属性値を変更し、次に地物のジオメトリを変更します。

```
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.fromPointXY(QgsPointXY(111,222))
    layer.dataProvider().changeGeometryValues({ fid : geom })
```

Tip: ジオメトリのみの編集には QgsVectorLayerEditUtils クラスをお勧めします

もし、ジオメトリを変更するだけなら、ジオメトリを編集（移動、頂点の挿入または移動など）するのに便利なメソッドを提供する QgsVectorLayerEditUtils を使うことを検討できるかもしれません。

6.4.4 ベクタ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を編集する場合、まず特定のレイヤの編集モードを開始し、次にいくつかの変更を行い、最後に変更をコミット（またはロールバック）する必要があります。コミットするまでは、変更内容はすべて書き込まれず、レイヤのインメモリ編集バッファに残ります。この機能はプログラムでも使用できます。これは、データプロバイダの直接利用を補完する、ベクタレイヤ編集の別の方法です。ベクタレイヤ編集用の GUI ツールを提供する場合、このオプションを使用します。これは、コミット/ロールバックするかどうかをユーザーが決定でき、元に戻す/やり直すが使用できるようになるからです。変更がコミットされると、編集バッファのすべての変更がデータプロバイダーに保存されます。

メソッドはすでに見たプロバイダにおけるものとよく似ていますが、プロバイダではなく QgsVectorLayer オブジェクトで呼び出されます。

これらのメソッドが機能するためには、そのレイヤは編集モードでなければいけません。編集モードを開始するには、 startEditing() メソッドを使用します。編集を終了するには、 commitChanges() メソッドか、もしくは rollBack() メソッドを使用します。前者はすべての変更をデータソースにコミットします。一方後者は変更をすべて破棄し、データソースには一切、手をつけません。

あるレイヤが編集モードかどうかを知るには、 isEditable() メソッドを使用してください。

では、これら編集メソッドの使用方法を示す実例をいくつか見てもらいます。

```
from qgis.PyQt.QtCore import QMetaType

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

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

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

# add new field
layer.addAttribute(QgsField("mytext", QMetaType.Type.QString))
```

(次のページに続く)

(前のページからの続き)

```
# remove a field
layer.deleteAttribute(fieldIndex)
```

取り消し／やり直しを適切に機能させるためには、上記のメソッド呼び出しを undo コマンドでラップしなければなりません。取り消し／やり直し機能が不要で、変更を即座に保存したい場合は、 **データプロバイダを使って編集** したほうが手軽でしょう。

取り消し機能を使用するには次のように行います。

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

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

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

# ... more editing ...

layer.endEditCommand()
```

`beginEditCommand()` メソッドは内部的に「アクティブな」コマンドを生成し、ベクタレイヤでその後に起こる変化を記録し続けます。 `endEditCommand()` メソッドの呼び出しによって、コマンドはアンドゥスタックにプッシュされ、ユーザーが GUI から取り消し／やり直しをすることができるようになります。変更の最中に何か不具合が生じたときは、 `destroyEditCommand()` メソッドによってコマンドは削除され、コマンドがアクティブな間に行われたすべての変更はロールバックされます。

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 `with edit(layer)` 文も使用できます。

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

これは最後に `commitChanges()` メソッドを自動的に呼び出します。もし何らかの例外が発生したときは、 `rollBack()` メソッドを呼び出してすべての変更をロールバックします。 `commitChanges()` メソッドの実行の最中に問題に遭遇したとき（メソッドが False を返したとき）は、 `QgsEditError` 例外を送出します。

6.4.5 フィールドを追加または削除する

フィールド（属性）を追加するには、フィールドの定義を配列で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を配列で渡すだけです。

```
from qgis.PyQt.QtCore import QMetaType

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

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

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

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

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

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()
```

Tip: `with` に基づくコマンドを使って変更を直接保存する

`with edit(layer):` を使うと、最後に `commitChanges()` を呼び出して自動的に変更がコミットされます。例外が発生した場合は、全ての変更を `rollBack()` します。**ベクタレイヤを編集バッファで修正する** を参照してください。

6.5 空間インデックスを使う

空間インデックスは、頻繁にベクタレイヤ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 ポイントを求める方法は、すべてのポイント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、もしくは索引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。

空間インデックスは、QGIS ベクタレイヤに対してデフォルトでは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：

- QgsSpatialIndex クラスを使用して空間インデックスを作成します：

```
index = QgsSpatialIndex()
```

- インデックスに地物を追加します --- インデックスは QgsFeature オブジェクトを受け取り、内部のデータ構造に追加します。オブジェクトは手動で作成するか、プロバイダの getFeatures() メソッドを過去に呼び出したときのものを使用することができます。

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

- 代わりに、一括読み込みを使用してレイヤのすべての地物を一度に読み込むことができます

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

- 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(QgsPointXY(25.4, 12.7), 5)

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

また、 QgsSpatialIndexKDBush 空間インデックスを使うこともできます。このインデックスは **標準** の QgsSpatialIndex と似ていますが:

- 単独のポイント地物 **だけ** をサポートします
- **静的** です（構築後にインデックスに地物を追加することはできません。）
- **とても高速です！**
- 追加の地物要求を必要とせず、元の地物のポイントを直接検索することができます
- 真の **距離ベース** の検索に対応しています。つまり検索点からある半径にあるすべての点を返します

6.6 QgsVectorLayerUtils クラス

QgsVectorLayerUtils クラスには、ベクタレイヤで使用できる非常に便利なメソッドがいくつかあります。

例えば、 createFeature() メソッドは 、各フィールドのすべての最終的な制約とデフォルト値を保持してベクタレイヤに追加される QgsFeature を準備します:

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

getValues() メソッドは、フィールドや式の値を素早く取得することができます:

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

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

6.7 ベクタレイヤを作る

ベクタレイヤデータセットを作るには幾つかの方法があります:

- QgsVectorFileWriter クラス: ディスクにベクタファイルを書き込むための便利なクラスです。ベクタレイヤ全体を保存する writeAsVectorFormatV3() への静的呼び出しか、このクラスのインスタンスを作成して継承された addFeature() への呼び出しのどちらかを使います。このクラスは、GDAL がサポートする全てのベクタ形式（GeoPackage、Shapefile、GeoJSON、KML、その他）をサポートします。
- QgsVectorLayer クラス: データプロバイダーをインスタンス化し、データソースのパス (url) を解釈してデータに接続しアクセスします。一時的なメモリベースのレイヤ (memory) を作成し、GDAL ベクタデータセット (ogr) やデータベース (postgres, spatialite, mysql, mssql) など (wfs, gpx, delimitedtext...) へ接続するために使用することが出来ます。

6.7.1 QgsVectorFileWriter のインスタンスから

```
# SaveVectorOptions contains many settings for the writer process
save_options = QgsVectorFileWriter.SaveVectorOptions()
transform_context = QgsProject.instance().transformContext()
# Write to a GeoPackage (default)
error = QgsVectorFileWriter.writeAsVectorFormatV3(layer,
```

(次のページに続く)

(前のページからの続き)

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

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

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

また、`FieldValueConverter` を使って、異なる形式と互換性を持つようにフィールドを変換できます。例えば、配列の変数型（Postgres など）をテキスト型に変換する場合は、以下のようにします:

```
LIST_FIELD_NAME = 'xxxx'

```

(次のページに続く)

(前のページからの続き)

```
class ESRIValueConverter(QgsVectorFileWriter.FieldValueConverter):

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

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

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

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

デスティネーション CRS を指定することもできます -- `QgsCoordinateReferenceSystem` の有効なインスタンスが第 4 パラメータとして渡された場合、レイヤはその CRS に変換されます。

有効なドライバ名については、 `supportedFiltersAndFormats()` メソッドを呼ぶか、‘ OGR のサポートフォーマットを参照してください`_ --- "Code" 列にドライバ名として値を渡す必要があります。

オプションとして、選択された地物のみをエクスポートするかどうか、作成時にさらにドライバ固有のオプションを渡すか、ライターに属性を作成しないように指示するかどうかを設定できます... その他にもたくさんの（オプション）パラメータがあります; 詳細は `QgsVectorFileWriter` のドキュメントを参照してください。

6.7.2 地物から直接

```
from qgis.PyQt.QtCore import QMetaType

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

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

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

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

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

# add a feature
fet = QgsFeature()

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

```

(次のページに続く)

(前のページからの続き)

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

6.7.3 QgsVectorLayer クラスのインスタンスから作成する

QgsVectorLayer クラスによってサポートされているすべてのデータプロバイダのうちから、ここではメモリレイヤに焦点をあてましょう。メモリプロバイダは主にプラグインやサードパーティ製アプリの開発者に使われることを意図しています。ディスクにデータを格納することをしないため、開発者はなんらかの一時的なレイヤのための手っ取り早いバックエンドとしてこれを使うことができます。

このプロバイダは属性フィールドの型として string、int、double をサポートします。

メモリプロバイダは空間インデックスもサポートしています。これはプロバイダの createSpatialIndex() 関数を呼び出すことによって有効になります。空間インデックスが作成されると、複数の地物にわたって行う処理を、より小さな領域内でより速く行うことができます。これはあらためて地物すべてを走査する必要がなく、指定された領域内のみを走査すればよいからです。

メモリプロバイダは QgsVectorLayer コンストラクタにプロバイダ文字列として "memory" を渡すと作ることができます。

コンストラクタはレイヤのジオメトリタイプを定義する URI も必要とします。これは "Point"、"LineString"、"Polygon"、"MultiPoint"、"MultiLineString"、"MultiPolygon"、``"None"``のうちのひとつです。

URI ではメモリプロバイダの座標参照系、属性フィールド、URI 内でのメモリプロバイダのインデックスも指定できます。構文は、

crs=definition
: 座標参照系を指定します。定義には QgsCoordinateReferenceSystem.createFromString() で受け入れられる形式のいずれかを使用できます。

index=yes
: プロバイダが空間インデックスを使うように指定します。

field=name:type(length,precision)
: レイヤの属性を指定します。属性は名前を持ち、オプションとして型 (integer, double, string)、長さ、および精度を持ちます。フィールドの定義は複数あってかまいません。

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、メモリプロバイダの作成と投入について説明しています

```
from qgis.PyQt.QtCore import QMetaType

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

(次のページに続く)

(前のページからの続き)

```
pr = vl.dataProvider()

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

# add a feature
fet = QgsFeature()
fet.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(10,10)))
fet.setAttributes(["Johnny", 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、全てうまくいったかどうか確認しましょう

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

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

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

6.8 ベクタレイヤの表現 (シンボロジ)

ベクタレイヤがレンダリングされるとき、データの表現はレイヤに関連付けられた **レンダラー** と **シンボル** によって決定されます。シンボルは地物の視覚的表現を処理するクラスで、レンダラはそれぞれの地物でどのシンボルが使われるかを決定します。

指定したレイヤのレンダラは、以下のように取得することができます:

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

この参照を利用して、少しだけ探索してみましょう:

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

```
Type: singleSymbol
```

QGIS のコアライブラリには、いくつかの既知のレンダラータイプが用意されています:

データ型	クラス	説明
singleSymbol	QgsSingleSymbolRen	単一シンボル。全ての地物を同じシンボルでレンダリングします
categorizedSymbol	QgsCategorizedSymb	カテゴリごとに違うシンボルを使って地物をレンダリングします
graduatedSymbol	QgsGraduatedSymbol	段階に分けられたシンボル。それぞれの範囲の値によって違うシンボルを使って地物をレンダリングします

カスタムレンダラータイプもあるかもしれないので、これらのタイプだけだと決めつけないようにしてください。アプリケーションの QgsRendererRegistry に問い合わせれば、現在利用できるレンダラーを調べることができます:

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

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

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

```
renderer.dump()
```

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

6.8.1 単一シンボルレンダラー

レンダリングに使用されるシンボルは symbol() メソッドで取得し、 setSymbol() メソッドで変更できます（C++開発者向け注意：レンダラーがシンボルを所有することになります。）

特定のベクタレイヤで使用されるシンボルは、 setSymbol() に該当するシンボルのインスタンスを渡して呼び出すことで変更できます。**ポイント**、**ライン**、**ポリゴン** レイヤのシンボルは、対応するクラス QgsMarkerSymbol, QgsLineSymbol, QgsFillSymbol の createSimple() 関数を呼び出して作成することができます。

createSimple() に渡す辞書は、シンボルのスタイルプロパティを設定します。

例えば、次のコード例のように setSymbol() に QgsMarkerSymbol を渡して呼び出し、特定の **ポイント** レイヤで使用するシンボルを置換することができます:

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

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。

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

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、次のサンプルコードに倣うことができます：

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

```
{'angle': '0', 'cap_style': 'square', 'color': '255,0,0,255,rgb:1,0,0,1', 'horizontal_
↪anchor_point': '1', 'joinstyle': 'bevel', 'name': 'square', 'offset': '0,0',
 'offset_map_unit_scale': '3x:0,0,0,0,0,0', 'offset_unit': 'MM', 'outline_color': '35,
↪35,35,255,rgb:0.13725490196078433,0.13725490196078433,0.13725490196078433,1',
↪'outline_style': 'solid', 'outline_width': '0', 'outline_width_map_unit_scale':
↪'3x:0,0,0,0,0,0', 'outline_width_unit': 'MM', 'scale_method': 'diameter', 'size': '2
↪', 'size_map_unit_scale': '3x:0,0,0,0,0,0', 'size_unit': 'MM', 'vertical_anchor_
↪point': '1'}
```

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

6.8.2 カテゴリ値シンボル・レンダラー

カテゴリ値レンダラーを使用する場合、分類に使用する属性を照会および設定できます: `classAttribute()` および `setClassAttribute()` メソッドを用います。

カテゴリ値のリストを取得する

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

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

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

ここで、 `value()` はカテゴリ間の識別に用いられる値、 `label()` はカテゴリ説明に用いられるテキスト、 `symbol()` は割り付けられたシンボルを返すメソッドです。

レンダラーは通常、カテゴリに使用したオリジナルのシンボルとカラーランプも保存します: sourceColorRamp() と sourceSymbol() のメソッドです。

6.8.3 連続値シンボルレンダラー

このレンダラーは先ほど扱ったカテゴリ・シンボル・レンダラー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。

レンダラーで使われている領域の多くの情報を見つけるには

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

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

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

ここでも classAttribute() （分類属性名を見つけるため)、sourceSymbol()、sourceColorRamp() メソッドを使うことができます。さらに、 mode() メソッドがあり、等間隔、分位点、その他の方法で範囲を作成する方法を決定します。

もし連続値シンボルレンダラー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.PyQt import QtGui

myVectorLayer = QgsVectorLayer("testdata/data/data.gpkg|layername=airports",
↪"Airports layer", "ogr")
myTargetField = 'scalerank'
myRangeList = []
myOpacity = 1
# Make our first symbol and range...
myMin = 0.0
myMax = 50.0
myLabel = 'Group 1'
```

(次のページに続く)

(前のページからの続き)

```
myColour = QtGui.QColor('#ffee00')
mySymbol1 = QgsSymbol.defaultSymbol(myVectorLayer.geometryType())
mySymbol1.setColor(myColour)
mySymbol1.setOpacity(myOpacity)
myRange1 = QgsRendererRange(myMin, myMax, mySymbol1, myLabel)
myRangeList.append(myRange1)
#now make another symbol and range...
myMin = 50.1
myMax = 100
myLabel = 'Group 2'
myColour = QtGui.QColor('#00eeff')
mySymbol2 = QgsSymbol.defaultSymbol(
     myVectorLayer.geometryType())
mySymbol2.setColor(myColour)
mySymbol2.setOpacity(myOpacity)
myRange2 = QgsRendererRange(myMin, myMax, mySymbol2, myLabel)
myRangeList.append(myRange2)
myRenderer = QgsGraduatedSymbolRenderer('', myRangeList)
myClassificationMethod = QgsApplication.classificationMethodRegistry().method(
↪"EqualInterval")
myRenderer.setClassificationMethod(myClassificationMethod)
myRenderer.setClassAttribute(myTargetField)

myVectorLayer.setRenderer(myRenderer)
```

6.8.4 シンボルを操作する

シンボルを表現するために、 QgsSymbol という基本クラスがあり、3 つの派生クラスがあります:

- QgsMarkerSymbol --- ポイント地物用
- QgsLineSymbol --- ライン地物用
- QgsFillSymbol --- ポリゴン地物用

すべてのシンボルは 1 つ以上のシンボルレイヤ（ QgsSymbolLayer から派生したクラス）で構成されています。シンボルレイヤは実際のレンダリングを行い、シンボルクラス自体はシンボルレイヤのコンテナとしてのみ機能します。

シンボルのインスタンス（レンダラーなど）があれば、それを探索することができます: type() メソッドは、それがマーカー、ライン、塗りつぶしシンボルであるかを示します。また、シンボルの簡単な説明を返す dump() メソッドがあります。シンボルレイヤのリストを取得するには:

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

(次のページに続く)

(前のページからの続き)

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

```
0: SimpleMarker
```

シンボルの色を調べるには color() メソッドを、色を変更するには setColor() を用います。マーカーシンボルでは、さらに size() と angle() でシンボルのサイズと回転を取得することができます。ラインシンボルの場合、 width() メソッドは、線の幅を返します。

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。

シンボルレイヤーの操作

前述したように、シンボルレイヤ (QgsSymbolLayer のサブクラス) は地物の外観を決定します。 一般的に使用されるいくつかの基本的なシンボルレイヤクラスがあります。新しいシンボルレイヤタイプを実装することで、地物がどのようにレンダリングされるかを任意にカスタマイズすることが可能です。layerType() メソッドはシンボルレイヤクラスを一意に特定します。基本的でデフォルトのものは SimpleMarker, SimpleLine, SimpleFill というシンボルレイヤクラスです。

シンボルレイヤクラスで作成できるシンボルレイヤの種類は、以下のコードで全て把握することができます:

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

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

QgsSymbolLayerRegistry 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するものです。

シンボルレイヤデータにアクセスするには、その properties() メソッドを使用し、外観を決定するプロパティのキー-バリュー辞書を返します。各シンボルレイヤのタイプは、使用するプロパティの特定のセットを持っています。さらに、汎用メソッドである color(), size(), angle() および width() とそのセッターも存在する。もちろん、サイズと角度はマーカーシンボルレイヤにのみ、幅はラインシンボルレイヤにのみ有効です。

カスタムシンボルレイヤタイプの作成

データをどうレンダリングするかをカスタマイズしたいと考えているとします。思うままに地物を描画する独自のシンボルレイヤクラスを作成できます。次の例は指定した半径で赤い円を描画するマーカーを示しています:

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

class FooSymbolLayer(QgsMarkerSymbolLayer):

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

  def layerType(self):
     return "FooMarker"

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

  def startRender(self, context):
    pass

  def stopRender(self, context):
      pass

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

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

`layerType()` メソッドはシンボルレイヤの名前を決定します。この名前は全てのシンボルレイヤの中で一意でなければなりません。`properties()` メソッドは属性の永続化のために使用されます。`clone()` メソッドは全ての属性が全く同じであるシンボルレイヤのコピーを返さなければなりません。最後にレンダリングメソッドです: `startRender()` は最初の地物を描画する前に呼ばれ、 `stopRender()` は描画が完了したら呼ばれ、 `renderPoint()` はれレンダリングのために呼ばれます。点の座標はすでに出力座標に変換されています。

ポリラインとポリゴンにとって、違いはレンダリング方法だけです: `renderPolyline()` はラインのリストを受け取り、 `renderPolygon()` は最初のパラメータとして外側のリングの点のリスト、第 2 のパラメー

タとして内側のリングのリスト（またはなし）を受け取ります。

普通はユーザー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ーは円の半径を設定できます。次のコードはそのようなウィジェットの実装となります:

```
from qgis.gui import QgsSymbolLayerWidget

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

        self.layer = None

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

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

    def symbolLayer(self):
        return self.layer

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

このウィジェットは、シンボルプロパティダイアログに埋め込むことができます。シンボルプロパティダイアログでシンボルレイヤタイプが選択されると、シンボルレイヤのインスタンスとシンボルレイヤウィジェットのインスタンスが作成されます。そして、 `setSymbolLayer()` メソッドを呼び、シンボルレイヤをウィジェットに割り当てます。そのメソッドの中で、ウィジェットはシンボルレイヤの属性を反映するために UI を更新する必要があります。`symbolLayer()` メソッドは、シンボルに使用するために、プロパティダイアログでシンボルレイヤを再度取得するために使用します。

属性が変わるたびに、ウィジェットは `changed()` シグナルを発して、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必要があります。

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。

シンボルレイヤーのメタデータを作る必要があるでしょう

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

class FooSymbolLayerMetadata(QgsSymbolLayerAbstractMetadata):

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

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

fslmetadata = FooSymbolLayerMetadata()
```

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

親クラスのコンストラクタにレイヤーのタイプ（レイヤーが返すものと同じ）とシンボルのタイプ（marker/line/fill）を渡す必要があります。`createSymbolLayer()` メソッドは *props* 辞書に指定された属性を持つシンボルレイヤーのインスタンスを作成する処理をします。そして、シンボルレイヤーの設定ウィジェットを返す `createSymbolLayerWidget()` メソッドがあります。

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。

6.8.5 カスタムレンダラーの作成

もし地物をレンダリングするためのシンボルをどう選択する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地物からランダムに一つ選ぶ簡単なカスタムレンダラーです

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


class RandomRenderer(QgsFeatureRenderer):
  def __init__(self, syms=None):
    super().__init__("RandomRenderer")
    self.syms = syms if syms else [
```

(次のページに続く)

(前のページからの続き)

```
      QgsSymbol.defaultSymbol(QgsWkbTypes.geometryType(QgsWkbTypes.Point)),
      QgsSymbol.defaultSymbol(QgsWkbTypes.geometryType(QgsWkbTypes.Point))
    ]

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

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

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

  def usedAttributes(self, context):
    return []

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

親クラスである QgsFeatureRenderer のコンストラクタはレンダラー名（レンダラー間でユニークでなければならない）を必要とします。symbolForFeature() メソッドは、特定の地物に対してどのシンボルを使用するかを決定します。 startRender() と stopRender() ではシンボルレダリングの初期化/最終化の処理に対応します。usedAttributes() メソッドは、レンダラーが存在すると予想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() メソッドは、レンダラーのコピーを返す必要があります。

シンボルレイヤーと同様に、レンダラーの設定用の GUI を付けることが可能です。これは QgsRendererWidget から派生したものでなければなりません。次のサンプルコードでは、ユーザが最初のシンボルを設定するためのボタンを作成しています

```
from qgis.gui import QgsRendererWidget, QgsColorButton


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

(次のページに続く)

(前のページからの続き)

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

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

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

コンストラクタはアクティブレイヤー (QgsVectorLayer <qgis.core.QgsVectorLayer>`)、グローバルスタイル (QgsStyle <qgis.core.QgsStyle>`) と現在のレンダラのインスタンスを受け取ります。レンダラーがない場合、またはレンダラーのタイプが異なる場合は、新しいレンダラーに置き換えられ、そうでない場合は、現在のレンダラー（必要なタイプを既に持っている）を使用します。ウィジェットのコンテンツは、レンダラーの現在の状態を示すように更新する必要があります。レンダラダイアログが受け入れられると、ウィジェットの `renderer()` メソッドが呼び出されて現在のレンダラーを取得します --- それがレイヤーに割り当てられることになります。

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

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

class RandomRendererMetadata(QgsRendererAbstractMetadata):

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

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

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

rrmetadata = RandomRendererMetadata()
```

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

シンボルレイヤーと同様に、抽象メタデータコンストラクタはレンダラー名、ユーザーから見える名前、オプションでレンダラーのアイコンの名前を待ち受けます。`createRenderer()` メソッドは `QDomElement` インスタンスを渡し、DOM ツリーからレンダラーの状態を復元するために使用することができます。`createRendererWidget()` メソッドは、設定ウィジェットを作成します。レンダラーに GUI が付属していない場合は、存在する必要はありませんし、`None` を返すこともできます。

アイコンをレンダラーに関連付けるには、`QgsRendererAbstractMetadata` コンストラクタで第 3 引数（オプション）として指定します --- RandomRendererMetadata `__init__()` 関数にある基底クラスのコンストラクタは次のようになります

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

アイコンはメタデータクラスの `setIcon()` メソッドを使って後で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もできますし、Qt resource （PyQt5 は Python 用の .qrc コンパイラを含んでいます）からも読み込むことができます。

6.9 より詳しいトピック

TODO:

- シンボルの作成や修正
- スタイルの操作 (`QgsStyle`)
- カラーランプの操作 (`QgsColorRamp`)
- シンボルレイヤーとレンダラーのレジストリを調べる方法

第7章　ジオメトリの操作

ヒント: pyqgis コンソールを使わない場合、このページにあるコードスニペットは次のインポートが必要です:

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

空間地物を表すポイント、ラインストリング、ポリゴンは一般的にジオメトリと呼ばれます。QGIS では、これらは `QgsGeometry` というクラスで表現されます。

ひとつのジオメトリが実際には単純な（シングルパート）ジオメトリの集合であることがあります。このようなジオメトリは、マルチパートジオメトリと呼ばれます。それが 1 種類の単純ジオメトリだけを含む場合は、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びます。例えば、複数の島からなる国は、マルチポリゴンとして表現できます。

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。レイヤーから地物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。

すべての可能なジオメトリの構成と関係の説明と仕様は、OGC Simple Feature Access Standards で高度な詳細が確認できます。

7.1 ジオメトリの構成

PyQGIS では、ジオメトリを作成するためのいくつかのオプションが用意されています:

- 座標から

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

  座標は QgsPoint クラスまたは QgsPointXY クラスを用いて与えられます。これらのクラスの違いは、 QgsPoint は M と Z の次元をサポートしていることです。

  ポリライン (Linestring) は、ポイントのリストで表現されます。

  ポリゴンは、線形のリング（すなわち閉じた線分）のリストで表されます。最初のリングは外側のリング（境界）であり、オプションで続くリングはポリゴンの穴となります。いくつかのプログラムとは異なり、QGIS はリングを閉じてくれるので、最初のポイントを最後のポイントとして複製する必要はありません。

  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

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

- well-known バイナリ（WKB）から

```
g = QgsGeometry()
wkb = bytes.fromhex("0101000000000000000000454000000000000001440")
g.fromWkb(wkb)

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

7.2 ジオメトリにアクセス

まず、ジオメトリタイプを調べます。使用するのは wkbType() というメソッドです。これは QgsWkbTypes.Type の列挙から値を返します。

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

代替案として、 type() メソッドを使うことができます。このメソッドは QgsWkbTypes.GeometryType 列挙から値を返します。

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

displayString() 関数を使用すると、人間が読めるジオメトリタイプを取得することができます。

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

また、ジオメトリがマルチパートであるかどうかを調べるヘルパー関数 isMultipart() も用意されています。

ジオメトリから情報を抽出するために、各ベクタ型に対応したアクセサ関数が用意されています。ここでは、これらのアクセサの使い方の例を紹介します:

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

注釈: タプル (x,y) は実際のタプルではなく、 QgsPoint オブジェクトであり、値は x() と y() というメソッドでアクセスできます。

マルチパートジオメトリについては、同様のアクセス関数があります： asMultiPoint(), asMultiPolyline() および asMultiPolygon()。

ジオメトリのタイプに関係なく、ジオメトリのすべてのパーツに対して反復処理を行うことが可能です。例

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

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

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

```
LineString (0 0, 10 10)
```

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

```
Point (11 12)
```

また、 QgsGeometry.parts() メソッドを使ってジオメトリの各パーツを修正することも可能です。

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

print(geom.asWkt())
```

```
MultiPoint ((-10334726.79314758814871311 -5360105.10101194866001606),(-10462133.
↪82917747274041176 -5217484.34365733992308378),(-10589398.51346861757338047 -5072020.
↪35880533326417208))
```

7.3 ジオメトリの述語と操作

QGIS では、ジオメトリ述語（contains(), intersects() , ...) や集合演算 (combine(), difference(), ...) などの高度なジオメトリ操作に GEOS ライブラリを使っています。 また、面積（ポリゴンの場合）や長さ（ポリゴンとラインの場合）のようなジオメトリのプロパティを計算することもできます。

ここでは、与えられたレイヤーの地物を繰り返し処理し、そのジオメトリに基づいて幾何学的な計算を実行する例を見てみましょう。以下のコードは、チュートリアル QGIS プロジェクトの countries レイヤーにある各国の面積と周囲長を計算し、表示するものです。

以下のコードは レイヤ が QgsVectorLayer オブジェクトで ポリゴン地物タイプを持っていると仮定します。

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

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

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

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

さて、これでジオメトリの面積と外周を計算し、印刷することができました。しかし、その値がおかしなことにすぐに気づくでしょう。これは、QgsGeometry クラスの area() と length() を使って計算した場合、面積や周長は CRS を考慮しないためです。より強力な面積と距離の計算を行うには、 QgsDistanceArea クラスを使用することができ、楕円体ベースの計算を行うことができます:

以下のコードは レイヤ が QgsVectorLayer オブジェクトで ポリゴン地物タイプを持っていると仮定します。

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

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

(次のページに続く)

(前のページからの続き)

```
5
6  # let's filter for countries that begin with Z, then get their features
7  query = '"name" LIKE \'Z%\''
8  features = layer.getFeatures(QgsFeatureRequest().setFilterExpression(query))
9
10 for f in features:
11   geom = f.geometry()
12   name = f.attribute('NAME')
13   print(name)
14   print("Perimeter (m):", d.measurePerimeter(geom))
15   print("Area (m2):", d.measureArea(geom))
16
17   # let's calculate and print the area again, but this time in square kilometers
18   print("Area (km2):", d.convertAreaMeasurement(d.measureArea(geom), QgsUnitTypes.
   ↪AreaSquareKilometers))
```

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

また、2 点間の距離を知りたい場合もあります。

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

QGIS に含まれているアルゴリズムの多くの例を見つけて、これらのメソッドをベクターデータを分析し変換するために使用できます。ここにそれらのコードのいくつかへのリンクを記載します。

- `QgsDistanceArea` クラス: Distance matrix algorithm による距離と領域の計算
- Lines to polygons algorithm

第8章　投影法サポート

ヒント: pyqgis コンソールを使わない場合、このページにあるコードスニペットは次のインポートが必要です:

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

8.1 座標参照系

座標参照系 (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は複数の異なる方法で作ることができます：

- CRS を ID によって指定する

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

```
True
```

QGIS は以下の形式の CRS 識別子をサポートしています：

- EPSG: <code> --- EPSG 機関によって割り当てられた ID - createFromOgcWms() で処理されます
- POSTGIS: <srid>---PostGIS データベースで使われる ID - createFromSrid() で処理されます
- INTERNAL: <srsid> --- QGIS の内部データベースで使われる ID - createFromSrsId() で処理されます
- PROJ: <proj> - createFromProj() で処理されます
- WKT: <wkt> - :meth:`createFromWkt() <qgis.core.QgsCoordinateReferenceSystem.createFromWkt>`で処理されます

プリフィックスが指定されない場合は WKT の定義が仮定されます。

- CRS を well-known テキスト（WKT）で指定する

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

```
True
```

- 無効な CRS を作成し、create* 関数のいずれかを使用して初期化します。以下の例では、投影法を初期化するために Proj 文字列を使用しています。

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

```
True
```

CRS の作成（つまりデータベースへの参照）が成功したかどうかをチェックするのが賢明です：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 は内部データベース srs.db から適切な値を検索する必要があることに注意してください。そのため、独立したアプリケーションを作成する場合は、QgsApplication.setPrefixPath() で正しくパスを設定する必要があり、さもないとデータベースの検索に失敗します。QGIS Python コンソールからコマンドを実行している場合や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：すべてがすでに設定されています。

空間参照系の情報にアクセスする：

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

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

出力：

```
QGIS CRS ID: 3452
PostGIS SRID: 4326
Description: WGS 84
Projection Acronym: longlat
Ellipsoid Acronym: EPSG:7030
Proj String: +proj=longlat +datum=WGS84 +no_defs
Is geographic: True
Map units: 6
```

8.2 CRS の変換

QgsCoordinateTransform クラスを使用することで、異なる空間参照系の間で変換を行うことができます。最も簡単な使用方法は、変換元と変換先の CRS を作成し、 QgsCoordinateTransform インスタンスをそれらと現在のプロジェクトで作成することです。そして transform() 関数を繰り返し呼び出して変換を行います。デフォルトでは順変換を行いますが、逆変換も可能です。

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

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

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

出力：

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

第9章　マップキャンバスを使う

ヒント: pyqgis コンソールを使わない場合、このページにあるコードスニペットは次のインポートが必要です:

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

from qgis.PyQt.QtCore import Qt, QRectF

from qgis.PyQt.QtWidgets import QMenu

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

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

マップキャンバスウィジェットは QGIS で最も重要なウィジェットでしょう。なぜなら、重ね合う地図レイヤから構成される地図を表示し、地図やレイヤの対話処理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によって決まるマップの一部を表示します。対話処理は **マップツール** の使用により行われます。マップツールには:パン、ズーム、レイヤの識別、測定、ベクタ編集などがあります。他のグラフィックプログラムと同様に、常に 1 つのツールがアクティブで、ユーザーは利用可能なツールの間で切り替えることができます。

マップキャンバスは qgis.gui モジュールの `QgsMapCanvas` クラスで実装されます。この実装は、Qt Graphics

View フレームワークをベースにしています。このフレームワークは一般的に、カスタムグラフィックアイテムが配置され、ユーザーがそれらを操作することができるサーフェスとビューを提供します。 ここでは、グラフィックシーン、ビュー、アイテムの概念を理解できるほど Qt に精通していることを前提とします。もしそうでなければ、フレームワークの概要 を読んでください。

マップがパン、ズームイン/アウト（またはリフレッシュをトリガーする他のアクション）されるたびに、マップは現在の範囲内で再レンダリングされます。レイヤは画像にレンダリングされ（ QgsMapRendererJob クラスを使用)、その画像はキャンバス上に表示されます。また、 QgsMapCanvas クラスはレンダリングされたマップの更新を制御します。背景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** がある場合があります。

典型的なマップキャンバスアイテムは、ラバーバンド（測定、ベクター編集などに使用）または頂点マーカーです。キャンバスアイテムは通常、マップツールの視覚的なフィードバックを与えるために使用されます。例えば、新しいポリゴンを作成するとき、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のキャンバスアイテムを作成します。全てのマップキャンバスアイテムは QgsMapCanvasItem のサブクラスで、基本的な QGraphicsItem オブジェクトにさらにいくつかの機能を追加しています。

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:

- map canvas --- 地図の可視化
- マップキャンバスアイテム -- マップキャンバスに表示される追加の項目
- マップツール -- マップキャンバスとの対話処理用

9.1 マップキャンバスを埋め込む

マップキャンバスは他の Qt ウィジェットと同様にウィジェットなので、作成し表示するだけで簡単に使用できます。

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

マップキャンバスを持つスタンドアロンウィンドウを作成します。また、既存のウィジェットやウィンドウに埋め込むこともできます。 .ui ファイルと Qt デザイナーを使用する場合は、フォーム上に QWidget を配置し、それを新しいクラスに昇格させます: クラス名として QgsMapCanvas を、ヘッダーファイルとして qgis.gui をセットしてください。 pyuic5 ユーティリティがこれを処理します。これは、キャンバスを埋め込むためのとても便利な方法です。もう一つの可能性は、マップキャンバスと他のウィジェット（メインウィンドウやダイアログの子ウィジェットとして）を構築し、レイアウトを作成するコードを手動で書くことです。

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)
```

（不思議に思うかもしれませんが、 Qt は PyQt.QtCore モジュールに由来し、 Qt.white は定義済みの QColor インスタンスの 1 つです。）

それでは、地図レイヤを追加していきましょう。まずレイヤを開いて、現在のプロジェクトに追加します。次に、キャンバスの範囲を設定し、キャンバスのレイヤのリストを設定します。

```
vlayer = QgsVectorLayer("testdata/data/data.gpkg|layername=airports", "Airports layer
↪", "ogr")
if not vlayer.isValid():
    print("Layer failed to load!")

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

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

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

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が表示されているはずです。

9.2 ラバーバンドと頂点マーカー

キャンバスで地図の上に追加データを表示するには、マップキャンバスアイテムを使います。カスタムのキャンバスアイテムクラスを作る（以下で説明します）ことも可能ですが、便利なキャンバスアイテムクラスが 2 つあります: QgsRubberBand でポリラインやポリゴンを、 QgsVertexMarker でポイントを描画できます。どちらもマップ座標で動作するため、キャンバスがパンやズームされたときにシェイプが自動的に移動/拡大縮小されます。

ポリラインを表示するには:

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

ポリゴンを表示するには

```
r = QgsRubberBand(canvas, QgsWkbTypes.PolygonGeometry)  # polygon
points = [[QgsPointXY(-100, 35), QgsPointXY(10, 50), QgsPointXY(120, 35)]]
r.setToGeometry(QgsGeometry.fromPolygonXY(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、ポリゴンの線状のリングを含有するリングのリストです：最初のリングは外側の境界であり、さらに（オプションの）リング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。一時的に隠す（そして再び表示する）には hide() と show() のコンボを使います。アイテムを完全に削除するには、そのアイテムを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の描画にも使えますが、 QgsVertexMarker クラスの方が適しています（ QgsRubberBand だと目的の点の周りに矩形を描くだけです）。

頂点マーカーはこのように使うことができます:

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

これにより、位置 **[10,45]** に赤い十字が描かれます。アイコンの種類、サイズ、色、ペン幅はカスタマイズすることができます。

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

頂点マーカーを一時的に隠したり、キャンバスから取り除いたりする場合は、ラバーバンドの場合と同じ方法を使用します。

9.3 キャンバスで地図ツールを使用する

以下の例では、マップキャンバスと、地図のパンニングとズームのための基本的な地図ツールを含むウィンドウを作成します。パンニングは QgsMapToolPan で行い、ズームイン/ズームアウトは QgsMapToolZoom インスタンスのペアで行います。アクションはチェック可能に設定されており、後からツールに割り当てられ、アクションのチェック済み/チェック解除状態の自動処理を可能にします -- 地図ツールがアクティブになると、そのアクションは選択されたと印が付き、前の地図ツールのアクションは選択解除されます。地図ツールは setMapTool() メソッドを使って起動します。

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

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

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

(次のページに続く)

(前のページからの続き)

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

        self.setCentralWidget(self.canvas)

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

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

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

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

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

        self.pan()

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

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

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

上記のコードを Python のコンソールエディターで試してみてください。キャンバスウィンドウを呼び出すには、次の行を追加して `MyWnd` クラスをインスタンス化します。これらの行は、現在選択されているレイヤを新しく作成されたキャンバス上にレンダリングします。

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

9.3.1 QgsMapToolIdentifyFeature を使って地物を選択します

マップツール QgsMapToolIdentifyFeature を使って、コールバック関数に送られる地物をユーザーに選択させることができます。

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

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

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

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

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

9.3.2 マップキャンバスのコンテクストメニューに項目を追加する

マップキャンバスとの対話処理は、 contextMenuAboutToShow シグナルを使って、コンテキストメニューに追加した項目からも行うことができます。

次のコードは、マップキャンバス上で右クリックしたときに、デフォルトのエントリーの横に *My menu* *My Action* アクションを追加します。

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

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

9.4 カスタム地図ツールを書く

カスタムツールを作成することで、ユーザーがキャンバス上で行ったアクションに対してカスタマイズした振る舞いを実装することができます。

マップツールは QgsMapTool クラスまたは派生クラスを継承し、既に見たように setMapTool() メソッドを用いてキャンバス上でアクティブツールとして選択しなければなりません。

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地図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, QgsWkbTypes.PolygonGeometry)
    self.rubberBand.setColor(Qt.red)
    self.rubberBand.setWidth(1)
    self.reset()

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

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

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

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

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

```

(次のページに続く)

(前のページからの続き)

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

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

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

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

      return QgsRectangle(self.startPoint, self.endPoint)

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

9.5 カスタムマップキャンバスアイテムを書く

ここでは、円を描くカスタムキャンバスアイテムの例を紹介します:

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

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

  def center(self):
```

(次のページに続く)

(前のページからの続き)

```
    return self.center

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

  def size(self):
    return self.size

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

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


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

第10章　地図のレンダリングと印刷

ヒント: このページのコードスニペットは、以下のインポートが必要です:

```
import os

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

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

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

入力データを地図として描画せねばならないときには、総じてふたつのアプローチがあります。 *QgsMapRendererJob* を使って手早く済ませるか、もしくは :class:`QgsLayout` クラスで地図を構成し、より精密に調整された出力を作成するかです。

10.1 単純なレンダリング

レンダリングは `QgsMapSettings` オブジェクトを生成してレンダリング設定を定義し、その設定で `QgsMapRendererJob` を生成します。後者が結果の画像を作成するために使用されます。

こちらが例です：

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

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

render = QgsMapRendererParallelJob(settings)

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

render.finished.connect(finished)

# Start the rendering
render.start()

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

10.2 異なる CRS を持つレイヤーをレンダリングする

レイヤが複数あり、それぞれの CRS が異なっている場合は、上記の単純な例ではおそらく求める結果は得られません。範囲計算から正しい値を得るためには、明示的に目的の CRS を設定する必要があります。

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

10.3 印刷レイアウトを使用して出力する

印刷レイアウトは、上に示した単純なレンダリングよりも洗練された出力を行いたい場合に非常に便利なツールです。マップビュー、ラベル、凡例、表、その他紙の地図に通常存在する要素で構成される複雑なマップレイアウトを作成することができます。このレイアウトは、PDF、SVG、ラスタ画像にエクスポートしたり、プリンターで直接印刷することができます。

レイアウトは多くのクラスで構成されています。これらはすべてコア・ライブラリに属しています。GUI ライブラリにはありませんが、QGIS アプリケーションには要素を配置するための便利な GUI があります。もしあなたが Qt Graphics View framework に馴染みがないのであれば、レイアウトはそれを基礎にしているので、今すぐドキュメントをチェックすることをお勧めします。

レイアウトの中心となるクラスは `QgsLayout` クラスで、Qt の QGraphicsScene クラスから派生したものです。インスタンスを作成してみましょう：

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

これはいくつかのデフォルト設定でレイアウトを初期化します。具体的には、レイアウトに空の A4 ページを追加します。`initializeDefaults()` メソッドを呼び出さずにレイアウトを作成することもできますが、レイアウトにページを追加する作業は自分で行う必要があります。

前のコードでは、GUI には表示されない「一時的な」レイアウトが作成されます。プロジェクト自体を変更することなく、また変更をユーザーに見せることなく、いくつかのアイテムを素早く追加してエクスポートする場合などに便利です。レイアウトをプロジェクトと一緒に保存／復元し、レイアウトマネージャで利用できるようにしたい場合は、次を追加してください：

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

これで、様々な要素（マップ、ラベル、...）をレイアウトに追加できるようになりました。これらのオブジェクトはすべて `QgsLayoutItem` クラスを継承したクラスで表現されます。

以下は、レイアウトに追加できる主なレイアウト項目の説明です。

- map --- ここにカスタムサイズのマップを作成し、現在のマップキャンバスをレンダリングします。

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

- label --- 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

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

- 凡例

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

- スケールバー

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

- ノードに基づく図形

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

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

props = {}
props["color"] = "green"
props["style"] = "solid"
props["style_border"] = "solid"
props["color_border"] = "black"
```

(次のページに続く)

(前のページからの続き)

```
props["width_border"] = "10.0"
props["joinstyle"] = "miter"

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

レイアウトに項目を追加したら、移動やサイズの変更ができます:

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

各項目の周囲にはデフォルトで枠が描かれます。それを取り除くには次のようにします:

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

レイアウト項目を手作業で作成する以外に、QGIS はレイアウトテンプレートをサポートしています。これは基本的に、すべての項目を.qpt ファイル（XML 構文）に保存した組版です。

その組版の準備ができたら（レイアウト項目が作られ、組版に追加されたら)、ラスタ出力やベクタ出力に進むことができます。

10.3.1 レイアウトの有効性をチェックする

レイアウトは相互に接続された項目の集合で構成され、修正中にこれらの接続が壊れてしまったり（削除されたマップに接続された凡例、ソースファイルが見つからない画像アイテムなど)、レイアウト項目にカスタム制約を適用したい場合があります。 QgsAbstractValidityCheck は、これを実現するのに役立ちます。

基本的なチェックはこのようなものです：

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

レイアウトマップ項目がウェブメルカトル図法に設定されるたびに警告を投げるチェックです：

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

(次のページに続く)

(前のページからの続き)

```
      res.title = 'Map projection is misleading'
      res.detailedDescription = 'The projection for the map item {} is set to <i>Web␣
↪Mercator (EPSG:3857)</i> which misrepresents areas and shapes. Consider using an␣
↪appropriate local projection instead.'.format(i.displayName())
      results.append(res)

  return results
```

さらに複雑な例を挙げます。レイアウトマップ項目に、そのマップ項目に表示されている範囲の外でしか有効でない CRS が設定された場合、警告が投げられます：

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

            if not bounds_crs.contains(i.extent()):
                res = QgsValidityCheckResult()
                res.type = QgsValidityCheckResult.Warning
                res.title = 'Map projection is incorrect'
                res.detailedDescription = 'The projection for the map item {} is set␣
↪to \'{}\', which is not valid for the area displayed within the map.'.format(i.
↪displayName(), i.crs().authid())
                results.append(res)

    return results
```

10.3.2 レイアウトをエクスポートする

レイアウトをエクスポートするには、`QgsLayoutExporter` クラスを使わなければなりません。

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

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

PDF ファイルの代わりに、個々の SVG 又は画像ファイルにエクスポートしたいときは、`exportToSvg()` 又は `exportToImage()` を使います。

10.3.3 地図帳をエクスポートする

地図帳オプションが設定され、有効になっているレイアウトから全てのページをエクスポートしたい場合、エクスポータ (QgsLayoutExporter) の atlas() メソッドを少し調整して使用する必要があります。以下の例では、ページを PNG 画像にエクスポートしています：

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

出力は、地図帳で設定された出力ファイル名式を使用して、ベースパスのフォルダに保存されることに注意してください。

第11章　式、フィルタ適用および値の算出

ヒント: pyqgis コンソールを使わない場合、このページにあるコードスニペットは次のインポートが必要です:

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

QGIS は、SQL に似た式の解析を一部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）または関数（スカラー値を返す）として評価することができます。利用できる関数の完全なリストについては、ユーザーマニュアルの vector_expressions を参照してください。

3 つの基本的な型がサポートされています。

- 数 --- 整数および小数。例. `123`, `3.14`
- 文字列 --- 一重引用符で囲む必要があります: `'hello world'`
- 列参照 --- 評価する際に、参照はフィールド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`
- 数学的関数: `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`

- 変換関数： `to_int` 、 `to_real` 、 `to_string` 、 `to_date`
- ジオメトリ関数: `$area`, `$length`
- ジオメトリ処理関数： `$x` 、 `$y` 、 `$geometry` 、 `num_geometries` 、 `centroid`

以下の述語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)
- 論理述語: `AND`, `OR`, `NOT`
- NULL 値チェック: `IS NULL`, `IS NOT NULL`

述語の例:

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

スカラー式の例:

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

11.1 式を構文解析する

与えられた式が正しくパースできるかどうかは、以下の例で示す方法で確認します。

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

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

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

11.2 式を評価する

式は、例えば地物をフィルタしたり、新しいフィールド値を計算するなど、異なったコンテクストで使うことができます。いずれの場合においても、式は評価されなければなりません。つまり式の値は、単純な算術式から集約式まで、指定された計算ステップを実行することによって計算されます。

11.2.1 基本的な式

この基本的な式は単純な算術演算を評価します：

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

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

式は比較にも使用でき、1 (True) か 0 (False) かを評価します

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

11.2.2 地物に関わる式

地物に関わる式を評価するためには、式が地物のフィールド値にアクセスできるようにするために、QgsExpressionContext オブジェクトを生成し、評価関数に渡さなければなりません。

以下の例は、"Column" という名前のフィールドを持つ地物を作り、この地物を式のコンテクストに加える方法を示しています。

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

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

以下の例は、ベクターレイヤのコンテクストにおいて、新しいフィールド値を計算するためにどのように式を使うかを示す、より完成された例です。

```
from qgis.PyQt.QtCore import QMetaType

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

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

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

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

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

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

(次のページに続く)

(前のページからの続き)

```
context = QgsExpressionContext()
context.appendScopes(QgsExpressionContextUtils.globalProjectLayerScopes(vl))

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

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

```
876.5
```

11.2.3 式を使ってレイヤをフィルタする

次の例はレイヤーをフィルタリングして述語に一致する任意の地物を返します。

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

layer.startEditing()

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

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

matches = 0
for f in layer.getFeatures(request):
   matches += 1

print(matches)
```

```
7
```

11.3 式エラーを扱う

式をパースする過程、もしくは式を評価する過程で、式に関連するエラーが生じる可能性があります。

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

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

第12章　設定の読み込みと保存

ヒント: pyqgis コンソールを使わない場合、このページにあるコードスニペットは次のインポートが必要です:

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

多くの場合、プラグインでいくつかの変数が保存されて、そのプラグインを次に実行したときにユーザーがそれらの変数を入力したり選択したりする必要がないようにすると便利です。

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。

セッティングにはいくつかの種類があります：

- **グローバル設定** --- 特定のマシンのユーザーにバインドされます。QGIS 自身、多くのグローバル設定を格納しており、例えば、メインウィンドウのサイズやデフォルトのスナップ許容範囲などです。設定は `QgsSettings` クラスを使って、例えば `setValue()` や `value()` メソッドで処理されます。

  ここでは、これらの方法がどのように使われるかの例を見ることができます。

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

def read():
  s = QgsSettings()
  mytext = s.value("myplugin/mytext", "default text")
  myint  = s.value("myplugin/myint", 123)
  myreal = s.value("myplugin/myreal", 2.71)
  nonexistent = s.value("myplugin/nonexistent", None)
  print(mytext)
```

(次のページに続く)

(前のページからの続き)

```
print(myint)
print(myreal)
print(nonexistent)
```

value() メソッドの 2 番目のパラメータはオプションで、渡された設定名に対して以前の値が設定されていない場合に返されるデフォルト値を指定します。

グローバル設定のデフォルト値を qgs_global_settings.ini ファイルによって事前に設定する方法の詳細については、 deploying_organization を参照してください。

- **プロジェクト設定** --- プロジェクト間で異なるため、プロジェクトファイルと関連付けられます。マップキャンバスの背景色や、目的の座標参照系 (CRS) などはその一例です。あるプロジェクトでは白い背景と WGS84 が適しているかもしれませんが、別のプロジェクトでは黄色い背景と UTM 投影法が適しているかもしれません。

  使用例を以下に示します。

```
proj = QgsProject.instance()

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

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

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

ご覧の通り、writeEntry()) メソッドは多くのデータ型（整数、文字列、リスト）で使用されますが、設定値を読み出すためのメソッドは複数存在し、データ型ごとに対応するメソッドを選択する必要があります。

- **マップレイヤ設定** --- これらの設定は、あるプロジェクト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。そのため、1 つのシェープファイルで 2 つのマップレイヤインスタンスを作成した場合、それらの設定は共有されません。設定はプロジェクトファイル内に保存されるため、ユーザーがプロジェクトを再度開くと、レイヤ関連の設定は再びそこに存在します。設定値は `customProperty()` メソッドで取得し、 `setCustomProperty()` メソッドで設定します。

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

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

第13章　ユーザーとコミュニケーションする

ヒント: pyqgis コンソールを使わない場合、このページにあるコードスニペットは次のインポートが必要です:

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

from qgis.gui import (
    QgsMessageBar,
)

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

このセクションでは、ユーザーインターフェイスにおいて一貫性を維持するためにユーザーとのコミュニケーション時に使うべき方法と要素をいくつか示します。

13.1 メッセージを表示する。QgsMessageBar クラス

メッセージボックスを使用するのはユーザー体験の見地からは良いアイデアではありません。警告/エラー用に小さな情報行を表示するには、たいてい QGIS メッセージバーが良い選択肢です。

QGIS インターフェイ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できます。

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

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

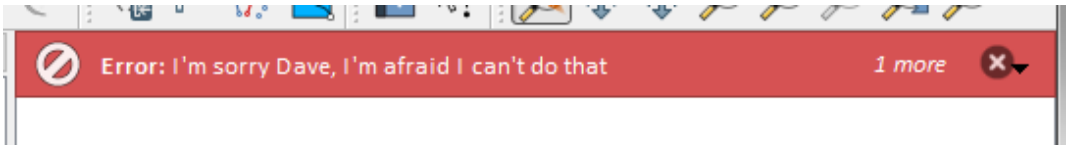


図 13.1: QGIS メッセージバー

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

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

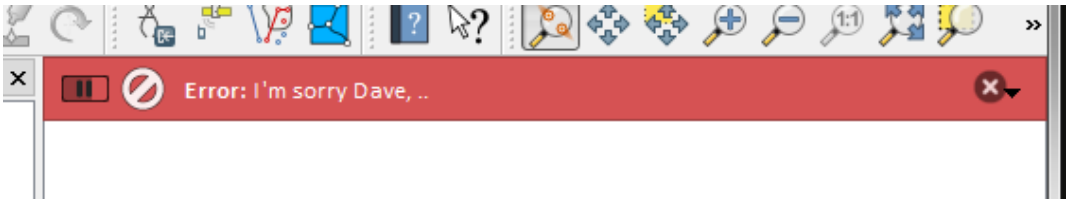


図 13.2: タイマー付き QGIS メッセージバー

上の例ではエラーバーを表示していますが、`level` パラメータは `Qgis.MessageLevel` 列挙型を使って警告メッセージや情報メッセージを作成することができます。最大 4 つのレベルを使用することができます：

0. Info
1. Warning
2. Critical
3. Success

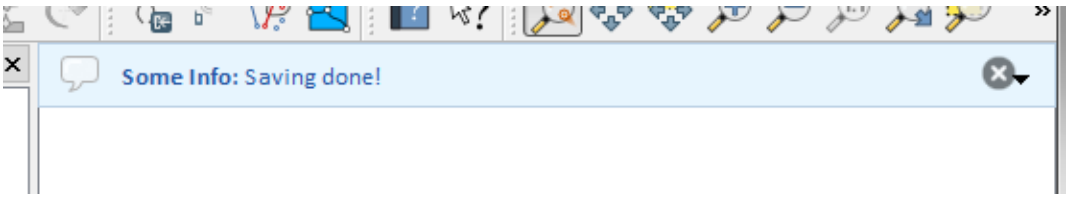


図 13.3: QGIS メッセージバー (お知らせ)

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

```
1 def showError():
2     pass
3
```

(次のページに続く)

(前のページからの続き)

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

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

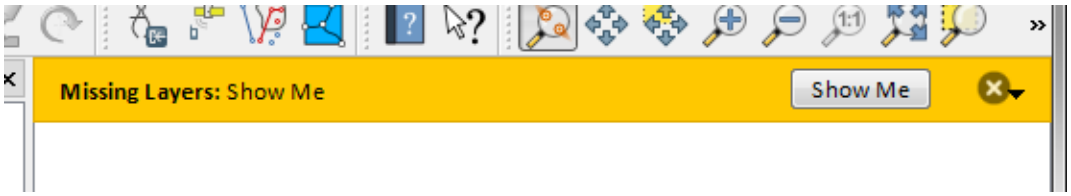


図 13.4: ボタン付きの QGIS メッセージバー

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=Qgis.Info)

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

図 13.5: カスタムダイアログ内の QGIS メッセージバー

13.2 プロセスを表示する

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

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

for i in range(10):
```

(次のページに続く)

(前のページからの続き)

```
    time.sleep(1)
    progress.setValue(i + 1)

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

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

また、次の例のように、内蔵のステータスバーを使って進捗状況を報告することもできます：

```
vlayer = iface.activeLayer()

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

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

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

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

13.3 ログ出力

QGIS では、コードの実行に関するすべての情報をログに記録して保存するために、3 つの異なるタイプのロギングが利用できます。それぞれ特定の出力場所があります。あなたの目的に合ったロギングの方法を使用することを検討してください：

- `QgsMessageLog` はユーザに問題を伝えるメッセージ向けです。QgsMessageLog の出力はログメッセージパネルに表示されます。
- python に組み込まれた **logging** モジュールは、QGIS Python API (PyQGIS) レベルでのデバッグ用です。Python スクリプト開発者が、地物 ID やジオメトリなどの Python コードをデバッグする必要がある場合にお勧めします。
- `QgsLogger` は、*QGIS* **内部** のデバッグ/開発者向けのメッセージです（壊れたコードによって何かが引き起こされた疑いがある）。メッセージは QGIS の開発者バージョンでのみ表示されます。

異なるロギングタイプの例を以下のセクションに示します。

警告: Python の print 文を使用することは、マルチスレッドされるコードでは安全ではなく、**アルゴリズムが極端に遅くなります**。これには、**式関数**、**レンダラー**、**シンボルレイヤ***、**プロセシングアルゴリズム** (特にその他) が含まれます。このような場合、代わりに python の **logging** モジュールかスレッドセーフなクラス（QgsLogger または QgsMessageLog）を使うべきです。

13.3.1 QgsMessageLog

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

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

注釈: QgsMessageLog の出力は、log_message_panel で見ることができます

13.3.2 python 内蔵のログモジュール

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

basicConfig メソッドは、ロギングの基本的な設定を行います。上記のコードでは、ファイル名、ロギングレベル、フォーマットが定義されています。ファイル名はログファイルを書き込む場所を指し、ロギングレベルは出力するレベルを定義し、フォーマットは各メッセージが出力されるフォーマットを定義します。

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

スクリプトを実行する度にログファイルを消去したいときは次のようにします：

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

python のロギング機能の使い方に関する更なるリソースは、以下を参照してください：

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

警告: ファイル名を設定してファイルにロギングしないと、ロギングがマルチスレッドになり、出力が著しく遅くなる可能性があることに注意してください。

第14章　認証インフラストラクチャ

ヒント: pyqgis コンソールを使わない場合、このページにあるコードスニペットは次のインポートが必要です:

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

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

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

from qgis.PyQt.QtNetwork import QSslCertificate
```

14.1 はじめに

認証基盤のユーザーリファレンスはユーザーマニュアル中で authentication_overview 段落を参照して下さい。

この章では、開発者の観点から、認証システムを使用するベストプラクティスについて説明します。

認証システムは、例えばレイヤが Postgres データベースへの接続を確立するときなど、特定のリソースへのアクセスに認証情報が必要なときにデータプロバイダーによって QGIS Desktop で広く使用されています。

また、QGIS の gui ライブラリには、プラグイン開発者が認証インフラをコードに簡単に統合するために使用できるウィジェットがいくつか用意されています:

- QgsAuthConfigEditor
- QgsAuthConfigSelect
- QgsAuthSettingsWidget

良いコードリファレンスは、認証基盤 tests code から読むことができます。

警告: 認証基盤の設計時に考慮されたセキュリティ上の制約により、内部メソッドの選択されたサブセットのみが Python に公開されます。

14.2 用語集

これはこの章で扱われる最も一般的なオブジェクトのいくつかの定義です。

マスターパスワード
: アクセスを許可し、QGIS 認証 DB に保存された資格情報を復号化するパスワードです

認証データベース
: A *Master Password* crypted sqlite db `qgis-auth.db` where *Authentication Configuration* are stored. e.g user/password, personal certificates and keys, Certificate Authorities

認証 DB
: 認証データベース

認証の設定
: 認証データの構成は 認証メソッド によって異なります。例えばベーシック認証の場合は、ユーザー/パスワードの対が格納されます。

認証設定
: 認証設定

認証方法
: 認証されるためには特別な方法が利用されています。各方法は、認証されるためにそれぞれ独自のプロトコルを利用しています。それぞれの方法は共有ライブラリとして QGIS 認証基盤の初期化中に動的にロードされるように実装されています。

14.3 エントリポイント QgsAuthManager

`QgsAuthManager` シングルトンは、QGIS の暗号化された *Authentication DB*（アクティブな user profile フォルダー下の `qgis-auth.db` ファイル）に格納されている認証情報を使用するエントリポイントです。

このクラスは、マスターパスワードの設定を求めたり、暗号化された保存情報にアクセスするためにマスターパスワードを透過的に使用したりすることで、ユーザーとの対話を行います。

14.3.1 マネージャを初期化し、マスターパスワードを設定する

次のコード例は、認証設定へのアクセスを開くために、マスターパスワードを設定する例を示します。コードのコメントは、このコード例を理解するために重要です。

```
authMgr = QgsApplication.authManager()

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

14.3.2 認証データベースに新しい認証構成項目を設定する

保存された資格情報は QgsAuthMethodConfig クラスの *Authentication Configuration* インスタンスで、次のような一意の文字列を使ってアクセスします:

```
authcfg = 'fm1s770'
```

この文字列は、QGIS API や GUI を使用してエントリーを作成する際に自動的に生成されますが、（異なる資格情報で）組織内の複数のユーザー間で設定を共有する必要がある場合、手動で既知の値に設定することが有用な場合があります。

QgsAuthMethodConfig は任意の *Authentication Method* のベースクラスです。任意の認証方法は、認証情報が格納されるコンフィギュレーションハッシュマップを設定します。以下は、仮想的な alice ユーザの PKI-path 認証情報を格納するための便利なスニペットです:

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

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

利用可能な認証方法

Authentication Method ライブラリは、認証マネージャの初期化時に動的にロードされます。利用可能な認証方式は以下の通りです:

1. Basic ユーザーとパスワード認証
2. EsriToken ESRI トークン型認証
3. Identity-Cert アイデンティティ証明書認証
4. OAuth2 OAuth2 認証
5. PKI-Paths PKI パス認証
6. PKI-PKCS # 12 PKI PKCS # 12 認証

認証局の導入

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

QgsPkiBundle で PKI バンドルを管理する

SslCert、SslKey、CA チェーンで構成される PKI バンドルをパックする便利なクラスは QgsPkiBundle クラスです。以下は、パスワードで保護されるようにするためのスニペットです:

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

バンドルから証明書/鍵/CAs を取り出すには、 QgsPkiBundle クラスドキュメントを参照してください。

14.3.3 authdb からエントリを削除する

次のスニペットは、authcfg 識別子を使って **認証データベース** からエントリを 削除します：

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

14.3.4 QgsAuthManager に authcfg 展開を残す

The best way to use an *Authentication Config* stored in the *Authentication DB* is referring it with the unique identifier `authcfg`. Expanding, means convert it from an identifier to a complete set of credentials. The best practice to use stored *Authentication Config*s, is to leave it managed automatically by the Authentication manager. The common use of a stored configuration is to connect to an authentication enabled service like a WMS or WFS or to a DB connection.

注釈: すべての QGIS データプロバイダーが認証インフラストラクチャに統合されているわけではないことを考慮してください。各認証メソッドは基本クラス `QgsAuthMethod` から派生し、異なるプロバイダのセットをサポートします。例えば、 `certIdentity()` メソッドは以下のプロバイダのリストをサポートしています:

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

サンプル出力:

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

例えば、 `authcfg = 'fm1s770'` で識別される、保存された資格情報を使って WMS サービスにアクセスするためには、次のコードのように、データ・ソースの URL に `authcfg` を使う必要があります。

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

上の場合には、 `wms` プロバイダーは、単に HTTP 接続を設定する前に、資格を `authcfg` URI パラメーターを拡張するために世話をします。

警告: 開発者は `authcfg` の拡張を `QgsAuthManager` に任せる必要があるでしょう。このようにすれば、拡張が早すぎることはないでしょう。

通常、データソースの設定には `QgsDataSourceURI` クラスを用いて構築された URI 文字列を以下のように使用することができます:

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

注釈: False パラメーターは、URI で authcfg ID の存在の URI 完全な展開を避けるために重要です。

他のデータプロバイダーと PKI の例

その他の例は、 test_authmanager_pki_ows や test_authmanager_pki_postgres というように QGIS テストアップストリームで直接読むことができます。

14.4 認証インフラストラクチャを使用するようにプラグインを適応させる

多くのサードパーティプラグインは、HTTP 接続を管理するために、 QgsNetworkAccessManager とそれに関連する認証基盤の統合の代わりに、httplib2 や他の Python ネットワークライブラリを使用しています。

この統合を容易にするために、NetworkAccessManager という Python のヘルパー関数が作られました。このコードは、ここ にあります。

このヘルパークラスは、次のスニペットのように使うことができます:

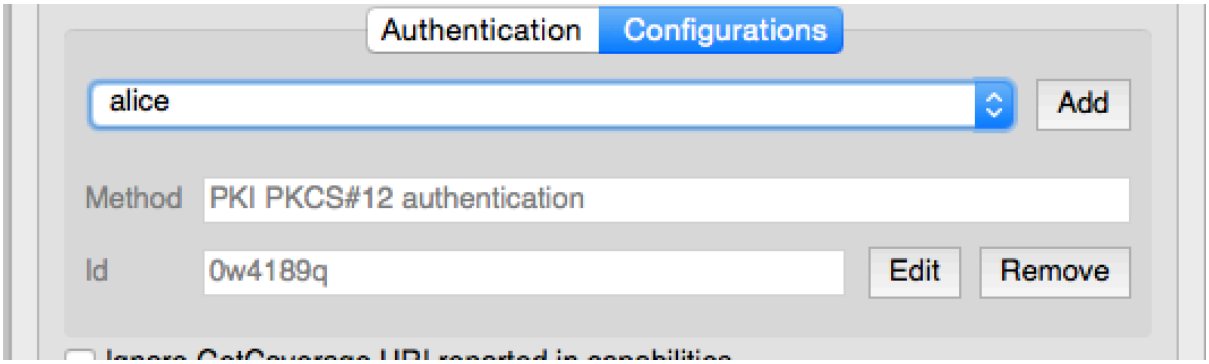
```
1 http = NetworkAccessManager(authid="my_authCfg", exception_class=My_
  ↪FailedRequestError)
2 try:
3   response, content = http.request( "my_rest_url" )
4 except My_FailedRequestError, e:
5   # Handle exception
6   pass
```

14.5 認証の GUI

この段落では、カスタムインターフェイスで認証インフラストラクチャを統合するために役立つ利用可能な GUI が記載されています。

14.5.1 資格情報を選択するための GUI

Authentication DB に格納されているセットから *Authentication Configuration* を選択する必要がある場合は、GUI クラス `QgsAuthConfigSelect` で利用可能です。



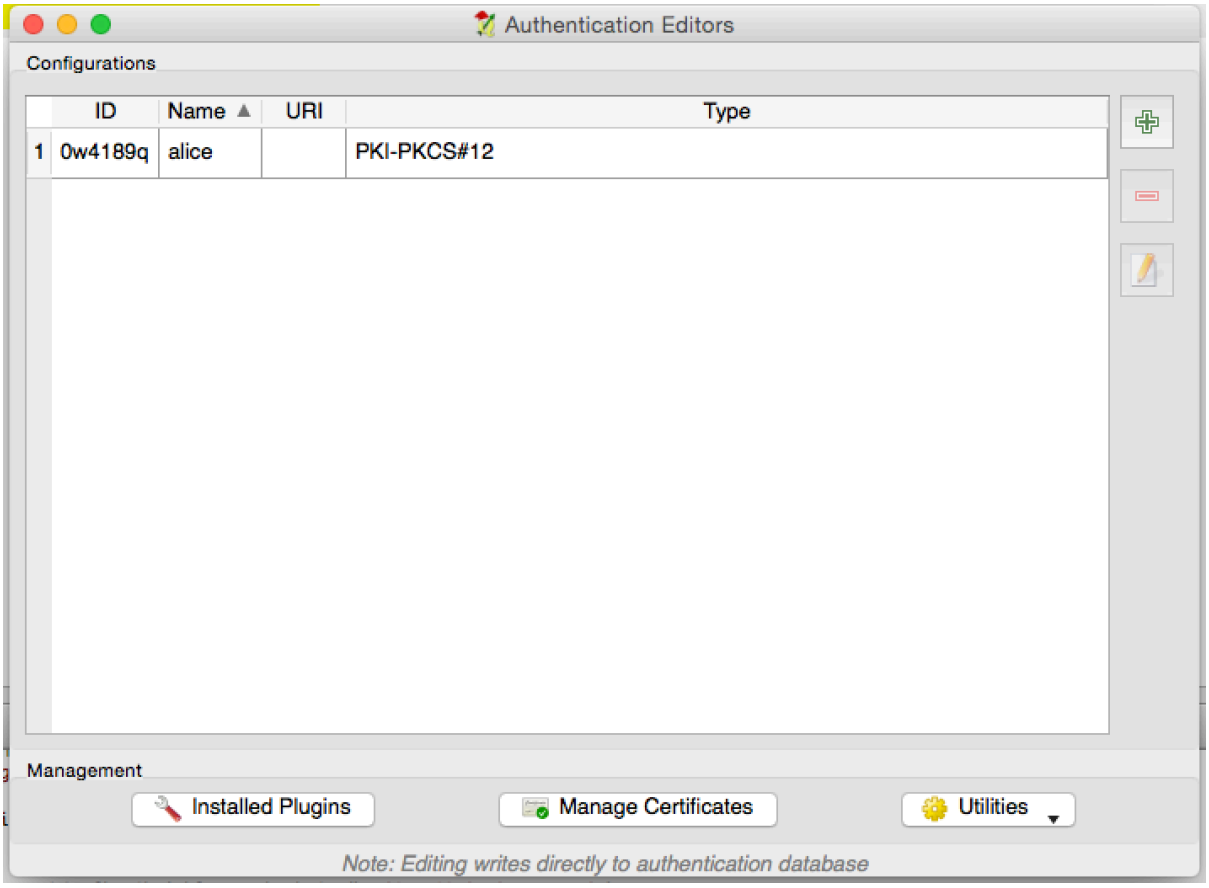
そして次のコードのように使用できます：

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

上記の例は、QGIS のソース code から引用しています。GUI コンストラクタの第 2 パラメータは、データプロバイダタイプを指します。このパラメータは、指定したプロバイダと互換性のある *Authentication Method* を制限するために使用されます。

14.5.2 認証エディタの GUI

資格情報、認証局を管理し、認証ユーティリティにアクセスするための完全な GUI は `QgsAuthEditorWidgets` クラスによって管理されています。



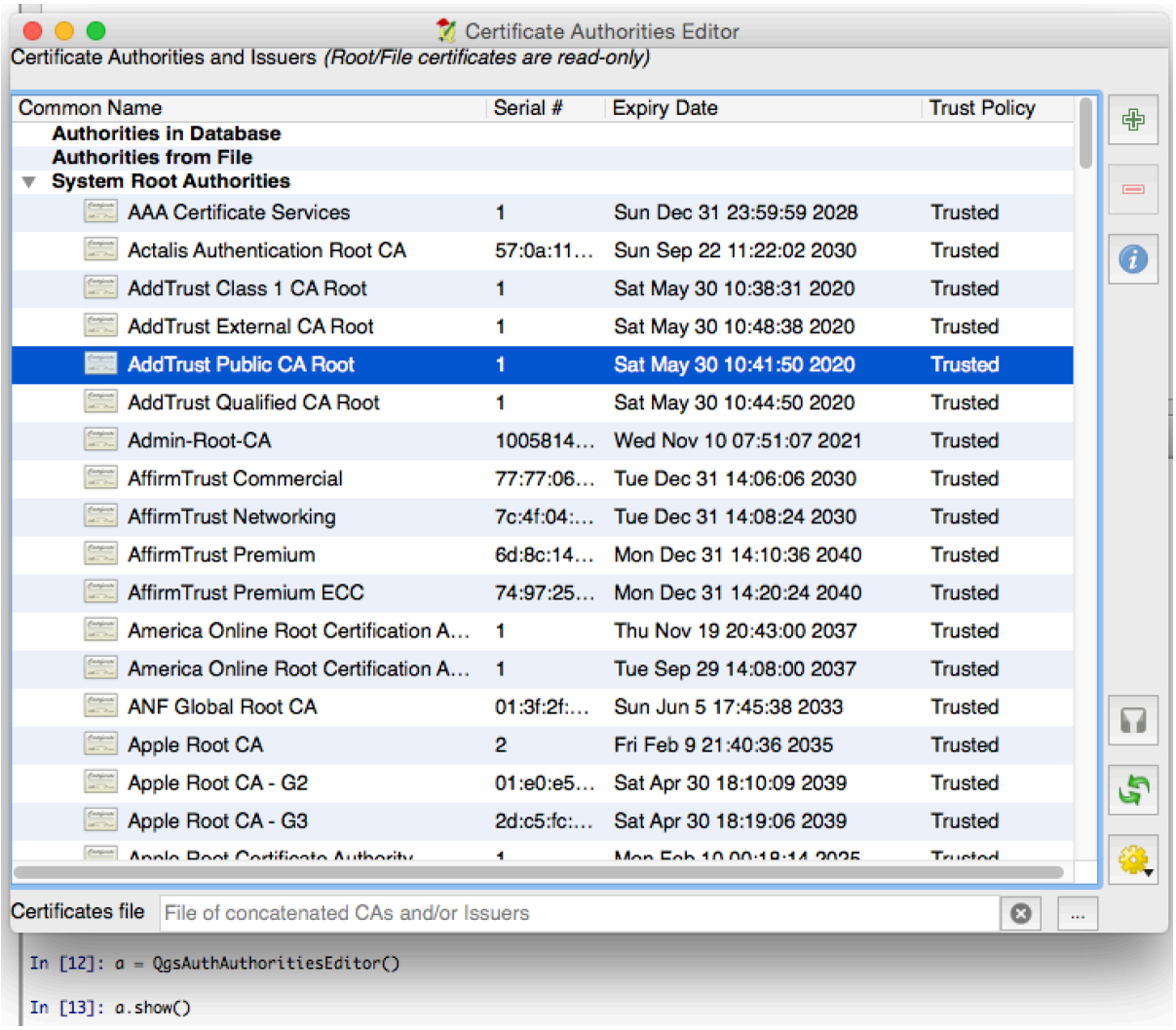
そして次のコードのように使用できます：

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

統合された例は、関連する test で見つけることができます。

14.5.3 認証局エディタの GUI

認証局だけを管理するための GUI は `QgsAuthAuthoritiesEditor` クラスで管理します。



そして次のコードのように使用できます：

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

第15章　タスク - バックグラウンドで重い仕事をする

ヒント: pyqgis コンソールを使わない場合、このページにあるコードスニペットは次のインポートが必要です:

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

15.1 はじめに

スレッドを使用したバックグラウンド処理は、重い処理が行われているときに応答性の高いユーザーインターフェイスを維持するための方法です。タスクは QGIS でスレッドを実行するために使用できます。

タスク（ `QgsTask` ）はバックグラウンドで実行されるコードのコンテナです。そしてタスクマネージャ（ `QgsTaskManager` ）は タスクの実行を制御するために使用されます。 これらのクラスは、シグナリング、進捗報告、およびバックグラウンドプロセスのステータスへアクセスするためのメカニズムを提供することによって、QGIS のバックグラウンド処理を単純化します。 タスクはサブタスクを使用してグループ化できます。

通常、グローバルタスクマネージャ（`QgsApplication.taskManager()` で見つかります）が使用されます。つまり、タスクマネージャによって制御されるタスクはあなたのタスクだけとは限りません。

QGIS タスクを作成する方法はいくつかあります：

- `QgsTask` を拡張することで自分のタスクを作成する

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

- 関数からタスクを作成

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

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

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

- プロセシングアルゴリズムからタスクを作成する

```
params = dict()
context = QgsProcessingContext()
context.setProject(QgsProject.instance())
feedback = QgsProcessingFeedback()

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

警告: どのバックグラウンドタスクも（どのように作成されたかに関わらず）、QgsVectorLayer や QgsProject へのアクセスや、新しいウィジェットの作成や既存のウィジェットとのインタラクションのような GUI ベースの操作のように、メインスレッドに存在する QObject を決して使ってはいけません。Qt ウィジェットへのアクセスや変更はメインスレッドからのみ行ってください。タスクで使用されるデータは、タスクが開始される前にコピーされなければなりません。バックグラウンドのスレッドからそれらを使用しようとするとクラッシュします。

さらに、常に `context` と `feedback` が少なくともそれらを使用するタスクと同じ期間生きていることを確実にしてください。タスクの完了時に、*QgsTaskManager* がタスクがスケジュールされた *context* と *feedback* にアクセスできない場合、QGIS はクラッシュします。

注釈: QgsProcessingContext`` を呼び出した直後に `setProject()` を呼び出すのが一般的なパターンです。これにより、タスクとそのコールバック関数がプロジェクト全体の設定のほとんどを使えるようになります。これは、コールバック関数で空間レイヤを扱うときに特に有用です。

タスク間の依存関係は `QgsTask` の `addSubTask()` 関数を使って記述することができます。依存関係が指定されると、タスクマネージャはこれらの依存関係がどのように実行されるかを自動的に決定します。可能な限り、依存関係はできるだけ早く満たすために並列に実行されます。他のタスクが依存するタスクがキャンセルされると、依存するタスクもキャンセルされます。循環依存関係はデッドロックを引き起こす

可能性があるので、注意が必要です。

タスクが利用可能なレイヤに依存している場合、 QgsTask の setDependentLayers() 関数を用いてそのことを指定することができます。タスクが依存するレイヤが利用できない場合、そのタスクはキャンセルされます。

タスクが作成されると、タスクマネージャの addTask() 関数を使用して実行をスケジュールすることができます。タスクをマネージャに追加すると、そのタスクの所有権は自動的にマネージャに移り、マネージャは実行後のタスクをクリーンアップして削除します。タスクのスケジューリングはタスクの優先度に影響されます。この優先度は addTask() で設定します。

タスクの状態は QgsTask および QgsTaskManager のシグナルと関数を使って監視できます。

15.2 例

15.2.1 QgsTask を拡張する

この例では RandomIntegerSumTask は QgsTask を拡張し、指定された期間中に 0 から 500 の間の 100 個のランダムな整数を生成します。 乱数が 42 の場合、タスクは中止され、例外が発生します。（サブタスク付きの） RandomIntegerSumTask のいくつかのインスタンスが生成されてタスクマネージャに追加され、2 種類の依存関係を実証します。

```
import random
from time import sleep

from qgis.core import (
    QgsApplication, QgsTask, QgsMessageLog, Qgis
    )

MESSAGE_CATEGORY = 'RandomIntegerSumTask'

class RandomIntegerSumTask(QgsTask):
    """This shows how to subclass QgsTask"""

    def __init__(self, description, duration):
        super().__init__(description, QgsTask.CanCancel)
        self.duration = duration
        self.total = 0
        self.iterations = 0
        self.exception = None

    def run(self):
        """Here you implement your heavy lifting.
        Should periodically test for isCanceled() to gracefully
        abort.
        This method MUST return True or False.
```

(次のページに続く)

(前のページからの続き)

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

    def finished(self, result):
        """
        This function is automatically called when the task has
        completed (successfully or not).
        You implement finished() to do whatever follow-up stuff
        should happen after the task is complete.
        finished is always called from the main thread, so it's safe
        to do GUI operations and raise Python exceptions here.
        result is the return value from self.run.
        """
        if result:
            QgsMessageLog.logMessage(
                'RandomTask "{name}" completed\n' \
                'RandomTotal: {total} (with {iterations} '\
              'iterations)'.format(
                    name=self.description(),
                    total=self.total,
                    iterations=self.iterations),
                MESSAGE_CATEGORY, Qgis.Success)
        else:
```

(次のページに続く)

(前のページからの続き)

```
            if self.exception is None:
                QgsMessageLog.logMessage(
                    'RandomTask "{name}" not successful but without '\
                    'exception (probably the task was manually '\
                    'canceled by the user)'.format(
                        name=self.description()),
                    MESSAGE_CATEGORY, Qgis.Warning)
            else:
                QgsMessageLog.logMessage(
                    'RandomTask "{name}" Exception: {exception}'.format(
                        name=self.description(),
                        exception=self.exception),
                    MESSAGE_CATEGORY, Qgis.Critical)
                raise self.exception

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


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

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

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

```
RandomIntegerSumTask(0): Started task "waste cpu subtask shortest"
```

(次のページに続く)

(前のページからの続き)

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

15.2.2 関数からのタスク

関数からタスクを作成します（この例では doSomething ）。 関数の最初のパラメータは関数の QgsTask を持ちます。 重要な（名前付き）パラメータは on_finished です。これはタスクが完了したときに呼ばれる関数を指定します。 この例の doSomething 関数は追加の名前付きパラメータ wait_time を持っています。

```
import random
from time import sleep

MESSAGE_CATEGORY = 'TaskFromFunction'

def doSomething(task, wait_time):
    """
    Raises an exception to abort the task.
    Returns a result if success.
    The result will be passed, together with the exception (None in
    the case of success), to the on_finished method.
    If there is an exception, there will be no result.
    """
    QgsMessageLog.logMessage('Started task {}'.format(task.description()),
                             MESSAGE_CATEGORY, Qgis.Info)
    wait_time = wait_time / 100
    total = 0
    iterations = 0
    for i in range(100):
        sleep(wait_time)
        # use task.setProgress to report progress
        task.setProgress(i)
        arandominteger = random.randint(0, 500)
        total += arandominteger
```

(次のページに続く)

(前のページからの続き)

```
        iterations += 1
        # check task.isCanceled() to handle cancellation
        if task.isCanceled():
            stopped(task)
            return None
        # raise an exception to abort the task
        if arandominteger == 42:
            raise Exception('bad value!')
    return {'total': total, 'iterations': iterations,
            'task': task.description()}

def stopped(task):
    QgsMessageLog.logMessage(
        'Task "{name}" was canceled'.format(
            name=task.description()),
        MESSAGE_CATEGORY, Qgis.Info)

def completed(exception, result=None):
    """This is called when doSomething is finished.
    Exception is not None if doSomething raises an exception.
    result is the return value of doSomething."""
    if exception is None:
        if result is None:
            QgsMessageLog.logMessage(
                'Completed with no exception and no result '\
                '(probably manually canceled by the user)',
                MESSAGE_CATEGORY, Qgis.Warning)
        else:
            QgsMessageLog.logMessage(
                'Task {name} completed\n'
                'Total: {total} ( with {iterations} '
                'iterations)'.format(
                    name=result['task'],
                    total=result['total'],
                    iterations=result['iterations']),
                MESSAGE_CATEGORY, Qgis.Info)
    else:
        QgsMessageLog.logMessage("Exception: {}".format(exception),
                                 MESSAGE_CATEGORY, Qgis.Critical)
        raise exception

# Create a few tasks
task1 = QgsTask.fromFunction('Waste cpu 1', doSomething,
                             on_finished=completed, wait_time=4)
task2 = QgsTask.fromFunction('Waste cpu 2', doSomething,
```

(次のページに続く)

(前のページからの続き)

```
                                on_finished=completed, wait_time=3)
QgsApplication.taskManager().addTask(task1)
QgsApplication.taskManager().addTask(task2)
```

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

15.2.3 プロセッシングアルゴリズムからのタスク

qgis:randompointsinextent アルゴリズムを使って、指定された範囲内に 50000 個のランダムな点を生成するタスクを作成します。 結果は安全な方法でプロジェクトに追加されます。

```
from functools import partial
from qgis.core import (QgsTaskManager, QgsMessageLog,
                       QgsProcessingAlgRunnerTask, QgsApplication,
                       QgsProcessingContext, QgsProcessingFeedback,
                       QgsProject)

MESSAGE_CATEGORY = 'AlgRunnerTask'

def task_finished(context, successful, results):
    if not successful:
        QgsMessageLog.logMessage('Task finished unsucessfully',
                                 MESSAGE_CATEGORY, Qgis.Warning)
    output_layer = context.getMapLayer(results['OUTPUT'])
    # because getMapLayer doesn't transfer ownership, the layer will
    # be deleted when context goes out of scope and you'll get a
    # crash.
    # takeMapLayer transfers ownership so it's then safe to add it
    # to the project and give the project ownership.
    if output_layer and output_layer.isValid():
        QgsProject.instance().addMapLayer(
             context.takeResultLayer(output_layer.id()))

alg = QgsApplication.processingRegistry().algorithmById(
                                     'qgis:randompointsinextent')
# `context` and `feedback` need to
# live for as least as long as `task`,
```

(次のページに続く)

(前のページからの続き)

```
# otherwise the program will crash.
# Initializing them globally is a sure way
# of avoiding this unfortunate situation.
context = QgsProcessingContext()
feedback = QgsProcessingFeedback()
params = {
    'EXTENT': '0.0,10.0,40,50 [EPSG:4326]',
    'MIN_DISTANCE': 0.0,
    'POINTS_NUMBER': 50000,
    'TARGET_CRS': 'EPSG:4326',
    'OUTPUT': 'memory:My random points'
}
task = QgsProcessingAlgRunnerTask(alg, params, context, feedback)
task.executed.connect(partial(task_finished, context))
QgsApplication.taskManager().addTask(task)
```

次も参照すること：https://www.opengis.ch/2018/06/22/threads-in-pyqgis3/。

第16章　Pythonプラグインを開発する

16.1 Python プラグインを構成する

プラグインを作成するための主な手順:

1. **考え**: 新しい QGIS プラグインで何をしたいのか、考えを持ちましょう。
2. **セットアップ**: **プラグインのファイルを作る**。プラグインの種類によって、必須のものもあれば、オプションのものもあります
3. **開発**: 適切なファイルに **コードを書く**
4. **ドキュメント**: **プラグインのドキュメントを書く**
5. オプションで: **翻訳**: 別の言語に **プラグインを翻訳する**
6. **テスト** ：すべてがうまく行くかどうかを **プラグインをリロード** して確認します。
7. **公開**：出来上がったプラグインを QGIS リポジトリに公開するか、ご自身のリポジトリを作って個人的な「GIS ウェポン」の兵器廠にしましょう。

16.1.1 はじめる

新しいプラグインを書き始める前に、 **公式な *Python* プラグインリポジトリ** を見てみてください。既存のプラグインのソースコードを見ることで、プログラミングについてより深く学ぶことができます。また、似たようなプラグインがすでに存在し、それを拡張したり、少なくともそれをベースにして自分のプラグインを開発することができるかもしれません。

プラグインのファイル構造をセットアップする

新しいプラグインを始めるには、必要なプラグインファイルをセットアップする必要があります。

始めるのに役立つ、プラグインテンプレートのリソースが 2 つあります:

- 教育目的や最小限のアプローチが必要な場合は、minimal plugin template が有効な QGIS Python プラグインを作成するために必要な基本的なファイル（スケルトン）を提供します。
- より完全な機能を持つプラグインテンプレートとして、Plugin Builder は、ローカライズ（翻訳）やテストなどの機能を含む、複数の異なるプラグインタイプのテンプレートを作成できます。

典型的なプラグインディレクトリには、以下のファイルを含みます:

- `metadata.txt` - **必須** - 一般的な情報、バージョン、名前、そしてプラグインのウェブサイトやプラグインインフラストラクチャで使用されるその他のメタデータが含まれています。
- `__init__.py` - **必須** - プラグインの開始点です。`classFactory()` メソッドを持つ必要があり、その他の初期化コードも持つことができます。
- `mainPlugin.py` - **コア・コード** - プラグインで主に働くコードです。プラグインのアクションに関するすべての情報とメインコードが含まれています。
- `form.ui` - **カスタム** *GUI* **を持つプラグイン用** - Qt Designer で作成した GUI です。
- `form.py` - **コンパイルした** *GUI* - 上記で説明した form.ui を Python に翻訳したものです。
- `resources.qrc` - **オプション** - Qt Designer によって作成される .xml ドキュメントです。GUI フォームで使用されるリソースへの相対パスが含まれています。
- `resources.py` - **コンパイルされたリソース、オプション** - 上記で説明した .qrc ファイルを Python に翻訳したものです。
- `LICENSE` - *required* if plugin is to be published or updated in the QGIS Plugins Directory, otherwise *optional*. File should be a plain text file with no file extension in the filename.

警告: プラグインを **公式な** *Python* **プラグインリポジトリ** にアップロードする場合は、プラグインが **検証** に必要ないくつかの追加ルールに従っているかどうかを確認する必要があります。

16.1.2 プラグインのコードを書く

上記で紹介した各ファイルに、どのような内容を追加していけばよいのか、次のセクションで紹介します。

metadata.txt

まず、プラグインマネージャはプラグインの名前、説明などの基本情報を取得する必要があります。この情報は `metadata.txt` に保存されます。

注釈: メタデータのエンコーディングはすべて UTF-8 でなければなりません。

メタデータ名	必須	注
name	Y	そのプラグインの名前から成る短い文字列
qgisMinimumVersion	Y	ドット記法による最小 QGIS バージョン
qgisMaximumVersion	N	ドット記法による最大 QGIS バージョン
description	Y	プラグインを説明する短いテキスト。HTML は不可
about	Y	プラグインの詳細を説明するより長いテキスト。HTML は不可
version	Y	バージョンをドット記法で記した短い文字列
author	Y	著者の名前
email	Y	著者の E メール。ウェブサイトでログインしたユーザのみに表示されるが、プラグインをインストールした後はプラグインマネージャで見ることができる
changelog	N	文字列。改行して複数行になってもよい。HTML 不可
experimental	N	ブール値フラグ、`True` または `False` - このバージョンが実験的であれば `True`
deprecated	N	ブール値のフラグ、`True` または `False`, アップロードされたバージョンだけでなく、プラグイン全体に適用されます
tags	N	コンマで区切ったリスト。個々のタグの内部ではスペース可
homepage	N	プラグインのホームページを示す有効な URL
repository	Y	ソースコードのリポジトリの有効な URL
tracker	N	チケットとバグ報告のための有効な URL
icon	N	画像のファイル名もしくは（プラグインの圧縮パッケージのベースフォルダからの）相対パス。画像はウェブに適したフォーマット（PNG, JPEG）で
category	N	`Raster`, `Vector`, `Database`, `Mesh` and `Web` のいずれか
plugin_dependencies	N	インストールする他のプラグインを PIP ライクなカンマ区切りでリストアップし、メタデータの name フィールドにあるプラグイン名を使います
server	N	ブール値のフラグ、`True` または `False` プラグインがサーバーインタフェースを持っているかどうかを決めます
hasProcessingProvider	N	ブール値のフラグ、`True` または `False`、プラグインがプロセシングアルゴリズムを提供するかどうかを決めます

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（プラグインのメニューエントリーを追加する方法は次のセクションで説明します）が、 **ラスタ**, **ベクタ**, **データベース**, **メッシュ** および *Web* メニューにも配置することができます。

表示するメニューを特定するために、対応する "category" メタデータエントリが存在します。これに従ってプラグインは分類することができます。このメタデータエントリは、ユーザへのヒントとして使われ、このプラグインを使うためにはどのメニューを探せばよいかを伝えます。"category" に使うことのできる値は、Vector、Raster、Database、Web です。例えば、あなたのプラグインを *Raster* メニューから使えるようにするには、 `metadata.txt` にそのように追加します。

```
category=Raster
```

注釈: *qgisMaximumVersion* が空欄の場合、**公式な** *Python* **プラグインリポジトリ** にアップロードする際に、自動的にメジャーバージョン＋ *.99* に設定されます。

metadata.txt のための例です。

```
; the next section is mandatory

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

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

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

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

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

(次のページに続く)

(前のページからの続き)

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

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

; Since QGIS 3.8, a comma separated list of plugins to be installed
; (or upgraded) can be specified.
; The example below will try to install (or upgrade) "MyOtherPlugin" version 1.12
; and any version of "YetAnotherPlugin".
; Both "MyOtherPlugin" and "YetAnotherPlugin" names come from their own metadata's
; name field
plugin_dependencies=MyOtherPlugin==1.12,YetAnotherPlugin
```

__init__.py

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。同時に QGIS はこのファイルの classFactory() 関数を必要とします。この関数はプラグインが QGIS に読み込まれる時に呼ばれます。この関数は QgisInterface ` のインスタンスへの参照を受け取り、 :file:`mainplugin.py ファイルのあなたのプラグインオブジェクトを返さなければなりません（ここでの事例では TestPlugin がそれに当たります。以下を見てください)。 __init__.py は次のようにならなければなりません。

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

# any other initialisation needed
```

mainPlugin.py

このファイルが魔法の起こる場所です。魔法はこんな風です（例えば mainPlugin.py の場合)。

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

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

class TestPlugin:

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

(次のページに続く)

(前のページからの続き)

```
    self.iface = iface

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

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

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

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

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

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

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

メインプラグインソースファイル（例えば `mainPlugin.py` ）に必須のプラグイン関数は以下の 3 つだけです。

- `__init__` は QGIS interface へのアクセスを与えます。
- `initGui()` はプラグインが読み込まれた時に呼ばれます。
- `unload()` はプラグインがアンロードされた時に呼ばれます。

上記の例では、 `addPluginToMenu()` が使われています。これにより、対応するメニューアクションが **プラグイン** メニューに追加されます。別のメニューにアクションを追加する別の方法も存在します。ここで

は、それらのメソッドの一覧を示します:

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

いずれも addPluginToMenu() メソッドと同じ構文になっています。

これらのあらかじめ定義されたメソッドのうちのひとつに、あなたのプラグインメニューを追加することは、プラグインエントリがどのように組織されているかの一貫性を保つためにも推奨されます。しかし、次の例が示すように、独自のカスタムメニューグループを直接メニューバーに追加することもできます。

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

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

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

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

カスタマイズが可能になるように、QAction クラスと QMenu クラスの objectName にあなたのプラグインの固有名を設定するのを忘れないでください。

ヘルプやアバウトのアクションはカスタムメニューに追加することもできますが、QGIS のメインメニューである **ヘルプ　プラグイン** メニューで利用できるようにすると便利です。これは pluginHelpMenu() メソッドを使って行われます。

```
def initGui(self):

    self.help_action = QAction(
        QIcon("testplug:icon.png"),
        self.tr("Test Plugin..."),
        self.iface.mainWindow()
```

(次のページに続く)

(前のページからの続き)

```
    )
    # Add the action to the Help menu
    self.iface.pluginHelpMenu().addAction(self.help_action)

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

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

def unload(self):

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

実際のプラグインを作成する場合は、別の（作業）ディレクトリでプラグインを書き、UI とリソースファイルを生成し、QGIS のインストールディレクトリにそのプラグインをインストールする makefile を書くのが賢明でしょう。

16.1.3 プラグインのドキュメントを書く

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することができます。`qgis.utils` モジュールは関数 `showPluginHelp()` を提供し、他の QGIS ヘルプと同じようにヘルプファイルのブラウザを開きます。

`showPluginHelp()` 関数は、呼び出したモジュールと同じディレクトリにある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` を順に探し、最初に見つかった方を表示します。ここで `ll_cc` は QGIS のロケールです。これにより、プラグインに複数の翻訳されたドキュメントを含めることができます。

`showPluginHelp()` 関数は、ヘルプを表示する特定のプラグインを特定する packageName、検索するファイル名の "index" と置き換える filename、ブラウザが置かれるドキュメント内の html アンカータグの名前 section をパラメータとして受け取ることもできます。

16.1.4 プラグインを翻訳する

いくつかのステップで、プラグインをローカライズするための環境を設定し、コンピュータのロケール設定によってプラグインが異なる言語で読み込まれるようにすることができます。

ソフトウェア要件

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

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

ファイルとディレクトリ

プラグインを作成すると、メインのプラグインディレクトリの中に i18n フォルダーが見つかります。

すべての翻訳ファイルは、このディレクトリ内になければなりません。

.pro ファイル

まず、.pro ファイルを作成します。これは、**Qt Linguist** で管理できる **プロジェクト** ファイルです。

この .pro ファイルでは、翻訳したいファイルやフォームをすべて指定する必要があります。このファイルは、ローカライズファイルと変数をセットアップするために使用されます。 *example plugin* の構造と一致する、プロジェクトファイル:

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

プラグインはもっと複雑な構造をしていて、複数のファイルにまたがる場合があります。この場合、.pro ファイルを読み込んで翻訳可能な文字列を更新するために使用するプログラムである pylupdate5 はワイルドカード文字を展開しないので、すべてのファイルを .pro ファイルに明示的に配置する必要があることに留意してください。そうすると、あなたのプロジェクトファイルは次のようなものになります:

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

さらに、your_plugin.py ファイルは、QGIS ツールバーのプラグインのすべてのメニューとサブメニューを **呼び出す** ファイルであり、それらをすべて翻訳したいのです。

最後に *TRANSLATIONS* 変数で、必要な翻訳言語を指定することができます。

> **警告:** ts ファイルの名前は必ず your_plugin_ + language + .ts のようにします。そうしないと言語の読み込みに失敗します！言語の 2 文字のショートカットを使用します（イタリア語は **it**、ドイツ語は **de** など...）

.ts file

.pro を作成したら、プラグインの言語用の .ts ファイルを生成する準備が整いました。

ターミナルを開き、your_plugin/i18n ディレクトリに移動し、次のように入力します:

```
pylupdate5 your_plugin.pro
```

your_plugin_language.ts ファイルが表示されます。

.ts ファイルを **Qt Linguist** で開き、翻訳を開始します。

.qm file

プラグインの翻訳が完了したら（一部の文字列が完成していない場合、その文字列のソース言語が使用されます)、.qm ファイル（QGIS で使用されるコンパイル済みの .ts ファイル）を作成する必要があります。

ターミナルを開き、your_plugin/i18n ディレクトリに cd し、次のように入力するだけです:

```
lrelease your_plugin.ts
```

これで、i18n ディレクトリに your_plugin.qm ファイルが表示されます。

Makefile を使って翻訳する

また、Plugin Builder でプラグインを作成した場合は、makefile を使って python コードや Qt ダイアログからメッセージを抽出することができます。Makefile の冒頭には、LOCALES 変数があります:

```
LOCALES = en
```

この変数に言語の略語を追加します。例えばハンガリー語は:

```
LOCALES = en hu
```

これで、ソースから hu.ts ファイル（と en.ts も）を生成したり更新したりすることができるようになりました:

```
make transup
```

これで、LOCALES 変数に設定されたすべての言語の .ts ファイルが更新されました。プログラムメッセージの翻訳には **Qt Linguist** を使用します。翻訳が終わると、transcompile: で .qm ファイルを作成することができます:

```
make transcompile
```

プラグインと一緒に .ts ファイルを配布する必要があります。

プラグインを読み込む

プラグインの翻訳を確認するには、QGIS を開き、言語を変更（**設定　オプション　一般**）して QGIS を再起動します。

プラグインが正しい言語で表示されるはずです。

> **警告:** プラグインに何か変更を加えた場合（新しい UI、新しいメニューなど）、`.ts` と `.qm` の両方のファイルの更新版を **再び生成する** 必要があるので、上記のコマンドを再度実行してください。

16.1.5 プラグインを共有する

QGIS は、プラグインリポジトリで何百ものプラグインをホストしています。あなたのプラグインを共有することを検討してください！それは QGIS の可能性を広げ、人々はあなたのコードから学ぶことができます。ホストされているすべてのプラグインは、QGIS のプラグインマネージャーから検索してインストールすることができます。

情報および要件はこちらです: plugins.qgis.org。

16.1.6 コツと技

プラグイン・リローダー

プラグインの開発中、テストのために QGIS に再読み込みする必要が頻繁に生じます。これは、**Plugin Reloader** プラグインを使用することで非常に簡単に行うことができます。このプラグインは プラグインマネージャ で見つけることができます。

qgis-plugin-ci でパッケージング、リリース、翻訳を自動化

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

CLI や CI アクションで、XML プラグインリポジトリファイルのリリース、翻訳、公開、生成を行うことができます。

プラグインにアクセスする

インストールされたプラグインの全てのクラスに QGIS 内から python を使ってアクセスすることができ、デバッグの際に便利です。

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

ログ・メッセージ

プラグインは log_message_panel 内に独自のタブを持ちます。

リソースファイル

プラグインによっては、例えば `resources.qrc` のように、アイコンなどの GUI 用のリソースを定義したリソースファイルを使用するものがあります:

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

他のプラグインや他の QGIS のパーツと衝突しないよう接頭辞を使うのは良いことです。さもないと望んでいないリソースを読み込んでしまうかもしれません。さてあと必要なのはこのリソースを含む Python ファイルを生成することです。これは **pyrcc5** コマンドを使って次のようにして行われます。

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

注釈: Windows 環境では、コマンドプロンプトや Powershell から **pyrcc5** を実行しようとすると、おそらく "Windows cannot access the specified device, path, or file [...]" というエラーが発生します。 最も簡単な解決策は OSGeo4W Shell を使用することですが、PATH 環境変数を変更したり、実行ファイルへのパスを明示的に指定することができる場合は、 `<Your QGIS Install Directory> \bin\pyrcc5.exe` に見つけることができるかもしれません。

16.2 コードスニペット

ヒント: pyqgis コンソールを使わない場合、このページにあるコードスニペットは次のインポートが必要です:

```
from qgis.core import (
    QgsProject,
    QgsApplication,
    QgsMapLayer,
)

from qgis.gui import (
    QgsGui,
    QgsOptionsWidgetFactory,
    QgsOptionsPageWidget,
    QgsLayerTreeEmbeddedWidgetProvider,
    QgsLayerTreeEmbeddedWidgetRegistry,
)

from qgis.PyQt.QtCore import Qt
from qgis.PyQt.QtWidgets import (
    QMessageBox,
    QAction,
    QHBoxLayout,
    QComboBox,
)
from qgis.PyQt.QtGui import QIcon
```

このセクションでは、プラグイン開発を容易にするコードスニペットを紹介します。

16.2.1 ショートカットキーでメソッドを呼び出す方法

プラグインに `initGui()` に追加します

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

`unload()` に追加します

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

CTRL+I が押されたときに呼び出されるメソッド

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

また、提供されたアクションのショートカットキーをユーザーがカスタマイズできるようにすることも可能です。これをするには次を加えます:

```
# in the initGui() function
QgsGui.shortcutsManager().registerAction(self.key_action)

# and in the unload() function
QgsGui.shortcutsManager().unregisterAction(self.key_action)
```

16.2.2 QGIS アイコンを再利用する方法

QGIS のアイコンはよく知られており、ユーザーに明確なメッセージを伝えることができるため、新しいアイコンを描画して設定する代わりに、プラグイン内で QGIS のアイコンを再利用したい場合があります。その場合は `getThemeIcon()` メソッドを使用してください。

例えば、QGIS コードリポジトリで入手できる  mActionFileOpen.svg アイコンを再利用するには：

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

`iconPath()` は QGIS アイコンを呼び出すもう一つの方法です。テーマアイコンの呼び出しの例は QGIS embedded images - Cheatsheet にあります。

16.2.3 オプションダイアログのプラグインのインタフェース

設定　オプション にカスタムプラグインオプションタブを追加することができます。QGIS アプリケーションの設定とプラグインの設定を 1 つの場所にまとめておくことで、ユーザーが発見しやすく誘導しやすくなるので、プラグインのオプションのために特定のメインメニュー項目を追加するよりも望ましい方法です。

以下のスニペットは、プラグインの設定のための新しい空白のタブを追加し、あなたのプラグインに固有のすべてのオプションと設定を入力できるようにします。以下のクラスを別のファイルに分割することができます。この例では、2 つのクラスをメインの `mainPlugin.py` ファイルに追加しています。

```
class MyPluginOptionsFactory(QgsOptionsWidgetFactory):

    def __init__(self):
        super().__init__()

    def icon(self):
        return QIcon('icons/my_plugin_icon.svg')

    def createWidget(self, parent):
        return ConfigOptionsPage(parent)


class ConfigOptionsPage(QgsOptionsPageWidget):

    def __init__(self, parent):
        super().__init__(parent)
        layout = QHBoxLayout()
        layout.setContentsMargins(0, 0, 0, 0)
        self.setLayout(layout)
```

最後に、インポートを追加して __init__ 関数を修正します：

```
from qgis.PyQt.QtWidgets import QHBoxLayout
from qgis.gui import QgsOptionsWidgetFactory, QgsOptionsPageWidget


class MyPlugin:
    """QGIS Plugin Implementation."""

    def __init__(self, iface):
        """Constructor.

        :param iface: An interface instance that will be passed to this class
            which provides the hook by which you can manipulate the QGIS
            application at run time.
        :type iface: QgsInterface
        """
        # Save reference to the QGIS interface
        self.iface = iface


    def initGui(self):
        self.options_factory = MyPluginOptionsFactory()
        self.options_factory.setTitle(self.tr('My Plugin'))
        iface.registerOptionsWidgetFactory(self.options_factory)

```

(次のページに続く)

(前のページからの続き)

```
    def unload(self):
        iface.unregisterOptionsWidgetFactory(self.options_factory)
```

Tip: レイヤプロパティダイアログにカスタムタブを追加する

レイヤプロパティダイアログにプラグインのカスタムオプションを追加するには、同様のロジックを適用して、クラス `QgsMapLayerConfigWidgetFactory` と `QgsMapLayerConfigWidget` を使うことができます。

16.2.4 レイヤツリーにレイヤのカスタムウィジェットを埋め込む

レイヤ パネルでレイヤエントリの隣や下に表示される通常のレイヤシンボロジの要素の他に、独自のウィジェットを追加することができ、レイヤでよく使用されるいくつかのアクションに素早くアクセスすることができます (フィルタリング、選択、スタイルの設定、ボタンウィジェットによるレイヤの更新、レイヤ型のタイムスライダーの作成、ラベルでの追加のレイヤ情報の表示... など)。これらの **レイヤツリー埋め込みウィジェット** は、個々のレイヤのプロパティ **凡例** タブから利用できます。

次のコードスニペットは、レイヤに利用可能なレイヤスタイルを表示するドロップダウンを凡例に作成し、異なるレイヤースタイルを素早く切り替えられるようにします。

```
class LayerStyleComboBox(QComboBox):
    def __init__(self, layer):
        QComboBox.__init__(self)
        self.layer = layer
        for style_name in layer.styleManager().styles():
            self.addItem(style_name)

        idx = self.findText(layer.styleManager().currentStyle())
        if idx != -1:
          self.setCurrentIndex(idx)

        self.currentIndexChanged.connect(self.on_current_changed)

    def on_current_changed(self, index):
        self.layer.styleManager().setCurrentStyle(self.itemText(index))

class LayerStyleWidgetProvider(QgsLayerTreeEmbeddedWidgetProvider):
    def __init__(self):
        QgsLayerTreeEmbeddedWidgetProvider.__init__(self)

    def id(self):
        return "style"

    def name(self):
```

(次のページに続く)

(前のページからの続き)

```
        return "Layer style chooser"

    def createWidget(self, layer, widgetIndex):
        return LayerStyleComboBox(layer)

    def supportsLayer(self, layer):
        return True   # any layer is fine

provider = LayerStyleWidgetProvider()
QgsGui.layerTreeEmbeddedWidgetRegistry().addProvider(provider)
```

レイヤの **凡例** プロパティタブから、レイヤスタイルチューザー を **使用可能なウィジェット** `から **使用するウィジェット** `にドラッグして、レイヤツリーでウィジェットを有効にします。埋め込まれたウィジェットは、関連するレイヤノードのサブアイテムの一番上に常に表示されます。

プラグインなどからウィジェットを使用したい場合は、次のように追加します：

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

16.3 プラグインを書いてデバッグするための IDE 設定

プログラマには皆それぞれ自分の好みの IDE /テキストエディタがありますが、ここに人気の IDE を設定し、QGIS の Python プラグインを書いたりデバッグするためのいくつかの推奨事項があります。

16.3.1 Python プラグインを書くのに便利なプラグイン

Python プラグインを書くときに便利なプラグインがあります。**プラグイン** **プラグインの管理とインストール...** から次をインストールします:

- *Plugin reloader*: これにより、QGIS を再起動することなくプラグインを再読み込みし、新しい変更を取り込むことができます。
- *First Aid*: これは、プラグインから例外が発生したときに変数を検査するための Python コンソールとローカルデバッガを追加します。

警告: 私たちは絶え間なく努力していますが、この行より先の情報は QGIS 3 用に更新されていない可能性があります。

16.3.2 Linux と Windows で IDE を設定する際の注意点

Linux では、通常、QGIS ライブラリの場所をユーザの PYTHONPATH 環境変数に追加するだけです。ほとんどのディストリビューションでは、~/.bashrc または ~/.bash-profile を以下の行で編集することでこれを行うことができます（OpenSUSE Tumbleweed でテスト済み）：

```
export PYTHONPATH="$PYTHONPATH:/usr/share/qgis/python/plugins:/usr/share/qgis/python"
```

ファイルを保存し、以下のシェルコマンドを使って環境設定を実行します：

```
source ~/.bashrc
```

Windowsでは、QGIS と同じ環境設定にし、QGIS と同じライブラリとインタプリタを使用する必要があります。一番手っ取り早い方法は、QGIS のスタートアップ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。 C:\OSGeo4W\bin\qgis-unstable.bat のようなものを探してください。

16.3.3 Pyscripter IDE を使ってデバッグする (Windows)

Pyscripter IDE を使うには、次のようにします：

1. qgis-unstable.bat のコピーを作成し、pyscripter.bat に名前を変更します。
2. それをエディタで開きます。そして最後の行、QGIS を起動する行を削除してください。
3. Pyscripter 実行ファイルを指す行を追加し、使用する Python のバージョンを設定するコマンドライン引数を追加します
4. 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ーを指す引数を追加します。これは OSGeoW インストールの bin フォルダー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
   ```

5. このバッチファイルをダブルクリックすると、正しいパスで Pyscripter が起動します。

Pyscripter よりも人気のある Eclipse は、開発者の間で一般的な選択肢です。次のセクションでは、プラグインの開発とテストのために Eclipse を設定する方法を説明します。

16.3.4 Eclipse と PyDev を使ってデバッグする

インストール

Eclipse を使うには、以下をインストールしてください

- Eclipse
- Aptana Studio 3 Plugin 又は PyDev
- QGIS 3.x
- QGIS のプラグインである **Remote Debug** をインストールすることもできます。現時点ではまだ実験的なプラグインなので、あらかじめ **プラグイン** **プラグインの管理とインストール...** **設定** で **実験的プラグインも表示** を有効にしてください。

Windows で Eclipse を使用する環境を準備するために、バッチファイルを作成し、それを使用して Eclipse を起動する必要があります：

1. `qgis_core.dll` が存在するフォルダーを探します。通常、これは `C:\OSGeo4W\apps\qgis\bin` ですが、自分の QGIS アプリケーションをコンパイルした場合、これはビルドフォルダの `output/bin/RelWithDebInfo` にあります
2. `eclipse.exe` 実行ファイルを探します。
3. 以下のスクリプトを作成し、QGIS プラグイン開発時に eclipse の起動に使います。

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

Eclipse をセットアップする

1. 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。

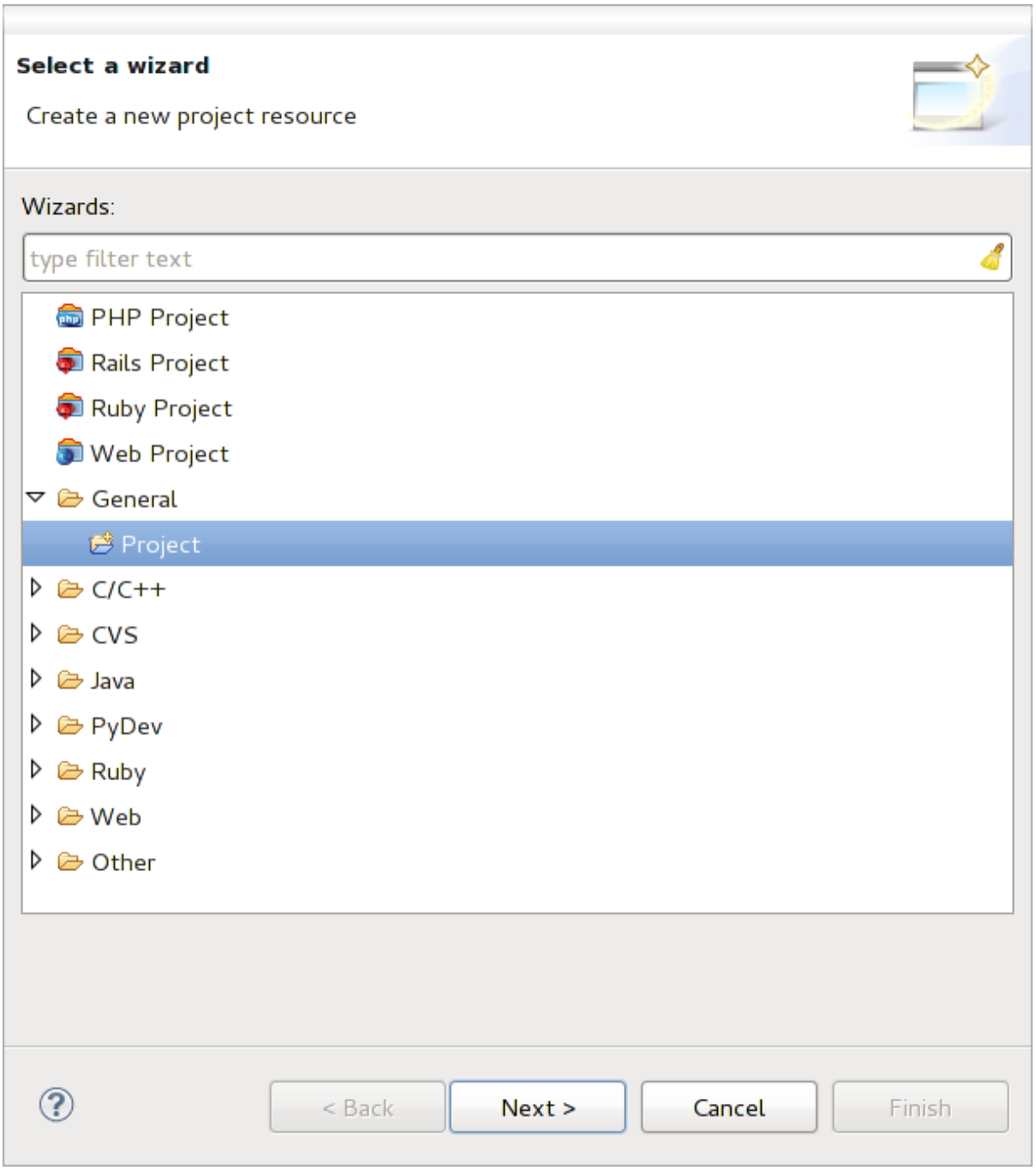


図 16.1: Eclipse プロジェクト

2. 新しいプロジェクトを右クリックして *New* *Folder* を選びます。
3. **詳細** をクリックし、 **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。

するとビュー **プロジェクトエクスプローラ** で、ソースツリーがポップアップしますので、コードでの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 ツールが使用できるようになっています。

デバッガを設定する

デバッガが動くようにするには:

1. Eclipse でデバッグパースペクティブに切り替えます（*Window* *Open Perspective* *Other* *Debug*）。
2. *PyDev* *Start Debug Server* を選んで PyDev デバッグサーバーを起動します。
3. 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。

ここでブレークポイントを設定できます。コードがそこに達すると実行は停止し、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す。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 )
```

図 16.2: 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。

1. コンソールビューを開きます（*Window* *Show view*）。あまり面白くない *Debug Server* コンソールが表示されます。しかし *Open Console* というボタンがあり、これを押すともっと面白い PyDev デバッグコンソールに切り替えることができます。
2. **コンソールを開く** ボタンの横にある矢印をクリックし、*PyDev* **コンソール** を選択します。どのコンソールを起動するかを尋ねるウィンドウが開きます。
3. *PyDev* **デバッグコンソール** を選択します。それがグレイアウトしていて、デバッガを開始して有効なフレームを選択するように指示された場合は、リモートデバッガが接続されていて、現在ブレークポイント上にあることを確認してください。

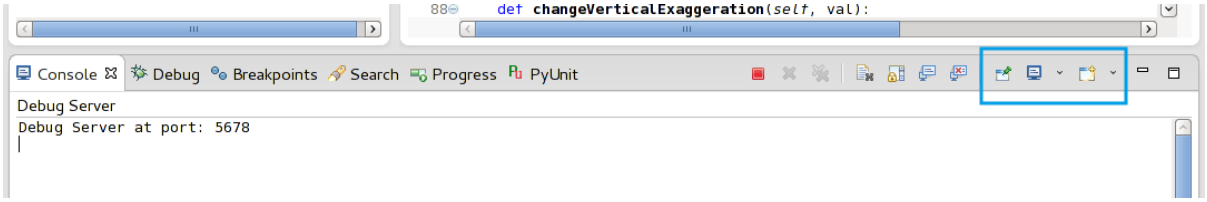


図 16.3: PyDev デバッグコンソール

これで、現在のコンテキストから任意のコマンドをテストできる対話型コンソールが完成しました。変数を操作したり、API を呼び出したり、好きなことができます。

Tip: 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デバッグサーバーに戻ります。この動作を停止するには、デバッグサーバーページで *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。

Eclipse に API を理解させる

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。

1. **ウィンドウ　設定**　*PyDev*　**インタプリタ**　*Python* をクリック。

   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** および **強制ビルトイン** です。

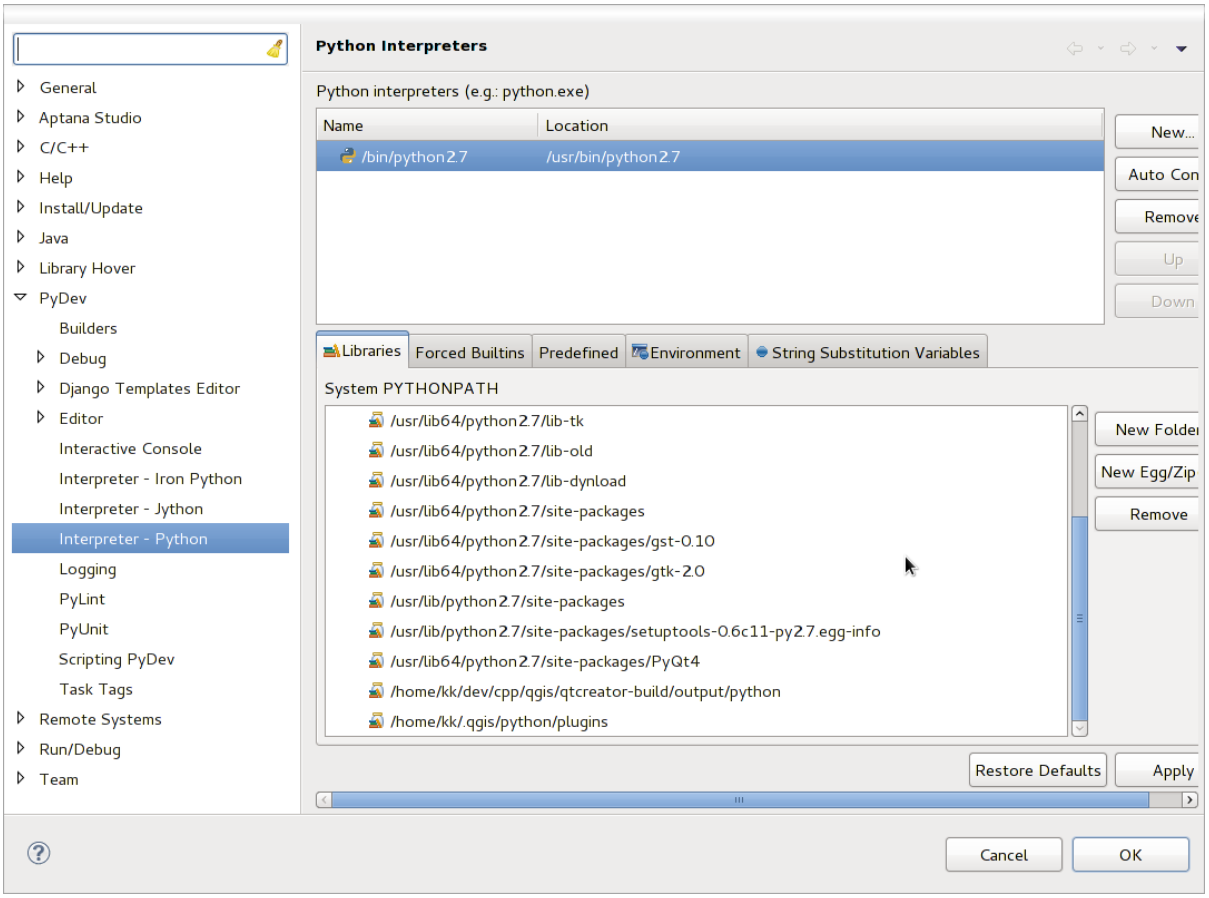


図 16.4: PyDev デバッグコンソール

2. 最初にライブラリタブを開きます。
3. 新しいフォルダーを追加し、QGIS のインストールにある python フォルダーを選択します。このフォルダーがどこにあるかわからない場合（plugins フォルダーではない場合）：
    1. QGIS を開く
    2. python コンソールを開始する
    3. `qgis` を入力する
    4. Enter を押す。どの QGIS モジュールを使用しているか、そのパスが表示されます。
    5. このパスから末尾の `/qgis/__init__.pyc` を取り除けば、目的のパスが得られます。
4. plugins フォルダーもここに追加してください（ user profile フォルダーの下の `python/plugins` にあります）。
5. 次に *Forced Builtins* タブにジャンプし、*New...* をクリックして `qgis` と入力します。これで Eclipse が QGIS API を解析するようになります。おそらく、Eclipse に PyQt API についても知ってもらいたいでしょう。そのため、PyQt を強制ビルトインとして追加します。おそらくライブラリタブにはすでに存在しているはずです。
6. *OK* をクリックして完了です。

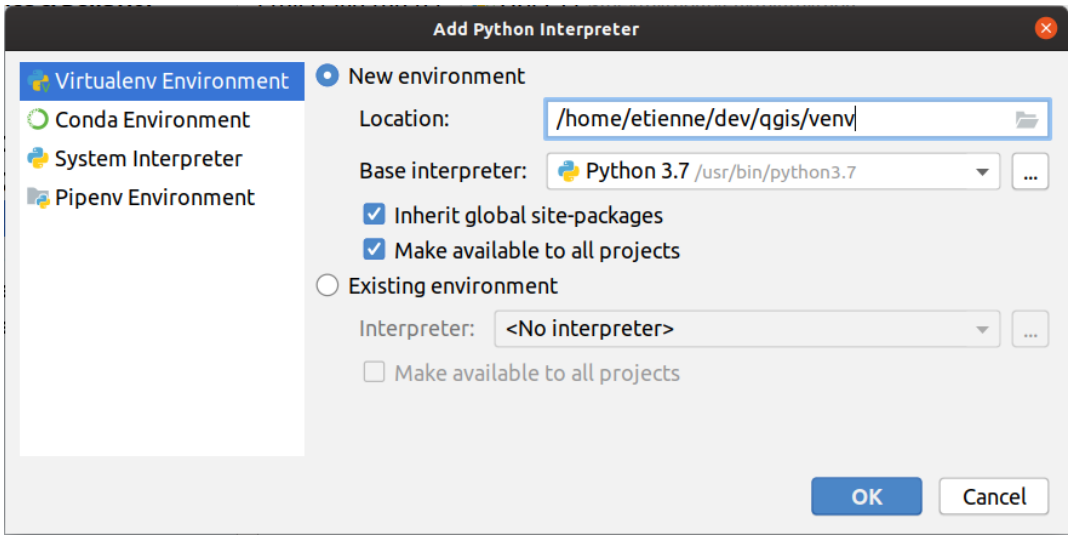
注釈: QGIS API が変更されるたびに（例えば QGIS master をコンパイルしていて SIP ファイルが変更された場合など）、このページに戻って *Apply* をクリックだけしてください。これで Eclipse がすべてのライブラリを再度解析します。

16.3.5 Ubuntu でコンパイルした QGIS を PyCharm でデバッグする

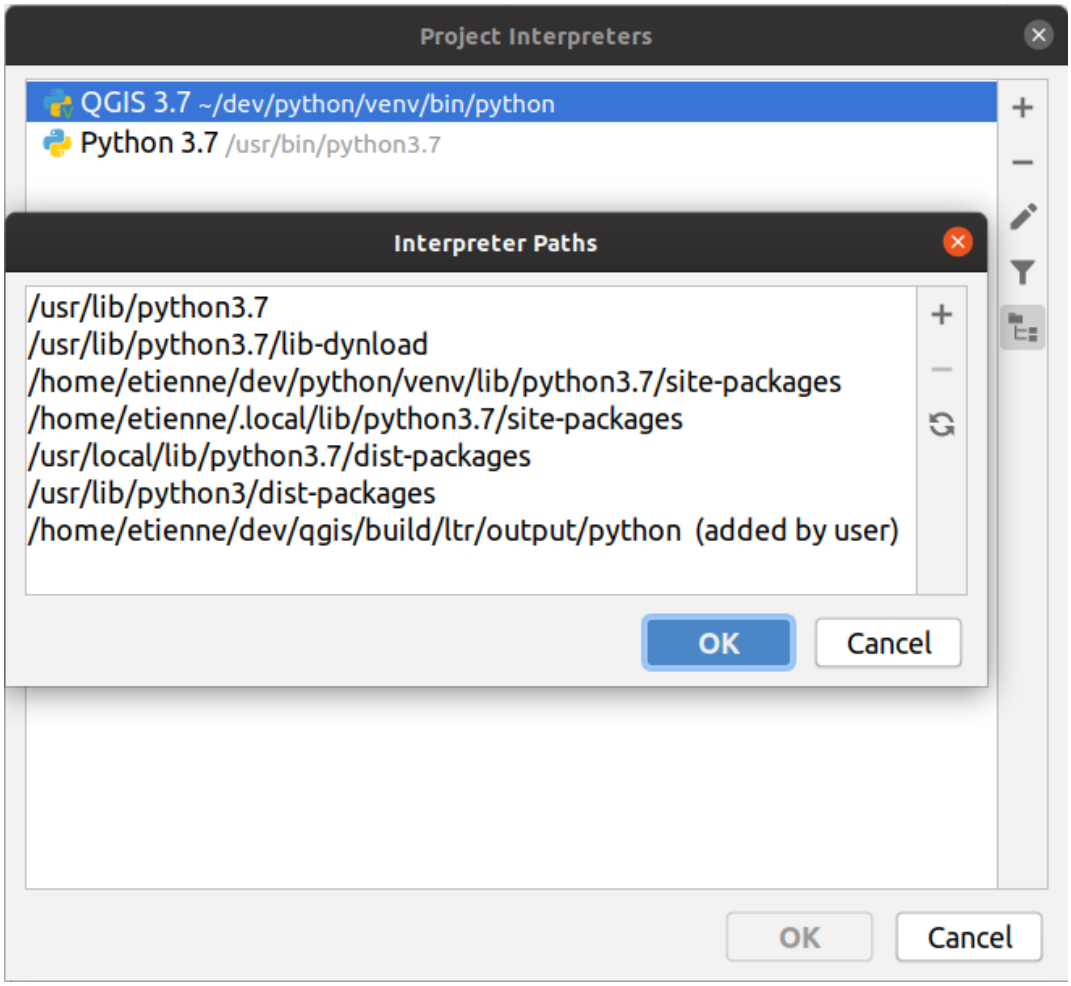
PyCharm は JetBrains 社が開発した Python 用の IDE です。Community Edition という無料版と Professional という有料版があります。PyCharm はウェブサイト https://www.jetbrains.com/pycharm/download からダウンロードできます

Ubuntu 上で指定されたビルドディレクトリ `~/dev/qgis/build/master` で QGIS をコンパイルしたと仮定します。自分でコンパイルした QGIS であることは必須ではありませんが、それでテストしています。パスは合わせる必要があります。

1. PyCharm の *Project Properties*, *Project Interpreter* に `QGIS` という Python 仮想環境を作成します。
2. 小さな歯車と次に *Add* をクリックします。
3. *Virtualenv environment* を選びます。
4. すべてのプラグインにこの Python インタプリタを使用するため、 `~/dev/qgis/venv` のように、すべての Python プロジェクトの一般的な場所を選択します。
5. あなたのシステムで利用可能な Python 3 ベースのインタプリタを選択し、次の 2 つのオプション、*Inherit global site-packages* と *Make available to all projects* をチェックしてください。



1. *OK* をクリックし、小さな歯車に戻り、:guilabel:`Show all`をクリックします。
2. 新しいウィンドウで、新しいインタプリタ `QGIS` を選択し、垂直メニューの最後のアイコン *Show paths for the selected interpreter.* をクリックします。
3. 最後に次の絶対パスをリストに加えます: `~/dev/qgis/build/master/output/python`



1. PyCharm を再起動すると、すべてのプラグインでこの新しい Python 仮想環境を使い始めることができます。

PyCharm は QGIS API を認識し、`from qgis.PyQt.QtCore import QDir` のように QGIS が提供する Qt を使用する場合は PyQt API も認識します。オートコンプリートが機能し、PyCharm がコードを検査できるようになります。

PyCharm のプロフェッショナル版では、リモートデバッグはうまく動作しています。コミュニティ版ではリモートデバッグは利用できません。ローカルデバッガにアクセスできるのは、QGIS 自身ではなく、PyCharm の **内部** で（スクリプトや unittest として）実行されるコードだけです。Python コードを *QGIS* **内で** 実行する場合は、前述の *First Aid* プラグインを使用するときでしょう。

16.3.6 PDB を利用してデバッグする

Eclipse や PyCharm のような IDE を使用していない場合は、以下の手順で PDB を使用してデバッグできます。

1. まず、デバッグしたい場所に次のコードを追加します。

```
# Use pdb for debugging
import pdb
```

(次のページに続く)

(前のページからの続き)

```
# also import pyqtRemoveInputHook
from qgis.PyQt.QtCore import pyqtRemoveInputHook
# These lines allow you to set a breakpoint in the app
pyqtRemoveInputHook()
pdb.set_trace()
```

2. 次に QGIS をコマンドラインから実行します。

   Linux ではこうします:

   ```
   $ ./Qgis
   ```

   macOS ではこうします:

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

3. そして、アプリケーションがブレークポイントに到達したら、コンソールに入力することができます！

TODO:
テストの情報を加える

16.4 プラグインをリリースする

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 **公式な** *Python* **プラグインリポジトリ** 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があります。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。

以下の提案には特に注意してください：

16.4.1 メタデータと名前

- 既存のプラグインに似すぎた名前は使わない
- あなたのプラグインが既存のプラグインと似たような機能を持っている場合、About フィールドで違いを説明してください。そうすれば利用者はインストールしてテストしなくても、どちらのプラグインを使えばよいかがわかります。
- プラグイン自身の名前で「プラグイン」を繰り返さない
- 1 行の説明にはメタデータの description フィールドを使い、より詳細な説明には About フィールドを使う。

- コードリポジトリ、バグトラッカー、およびホーム・ページを含めてください。これは非常に協働作業の可能性を向上させますし、利用可能な Web インフラストラクチャの 1 つ（GitHub、GitLab、Bitbucket など）で非常に簡単に行うことができます
- タグは注意して選択してください：情報価値のないもの（例：ベクタ）を避け、すでに他のユーザーによって使用されているもの（プラグインの Web サイトを参照）を選んでください
- 適切なアイコンを追加し、デフォルトのアイコンのままにしない；使用されるスタイルの提案については、QGIS インタフェースを参照してください

16.4.2 コードとヘルプ

- 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...) や無駄なもの (例えば.gitignore) をリポジトリにインクルードしない
- プラグインを適切なメニュー（ベクタ、ラスタ、Web、データベース）に加える
- 適切な場合（解析を実行するプラグイン）、プロセシングフレームワークのサブプラグインとしてプラグインを追加することを検討してください：これによってユーザーはバッチでそれを実行し、より複雑なワークフローにそれを統合できるようになり、インタフェースを設計する負担がなくなります
- 最低限の文書を、そして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。

16.4.3 公式な Python プラグインリポジトリ

公式 の Python プラグインのリポジトリは https://plugins.qgis.org/ にあります。

公式リポジトリを使用するには、OSGEO web portal から OSGEO ID を取得する必要があります。

プラグインをアップロードすると、スタッフによって承認され、あなたに通知されます。

TODO:
ガバナンス文書へのリンクを挿入する

許可

これらのルールは公式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ーが新しいプラグインを追加できます
- **スタッフ** ユーザーは、すべてのプラグインのバージョンを承認または非承認することができます
- 特別な権限 *plugins.can_approve* を持っているユーザーは、アップロードしたバージョンが自動的に承認されます
- 特別な権限 *plugins.can_approve* を持っているユーザーは、プラグインの **所有者** のリストにいる限り、他のユーザーがアップロードしたバージョンを承認することができます
- 特定のプラグインを削除および編集できるのは、**スタッフ** ユーザーとプラグインの **所有者** だけです

- *plugins.can_approve* 権限を持たないユーザーが新しいバージョンをアップロードした場合、そのプラグインのバージョンは自動的に未承認になります。

信頼マネジメント

スタッフはフロントエンドのアプリケーションから *plugins.can_approve* 権限を設定することで、選択したプラグイン作成者に **信頼** を与えることができます。

プラグインの詳細表示では、プラグイン作成者またはプラグイン **所有者** に信頼を与えるための直接リンクを提供します。

検証

プラグインのメタデータは、プラグインのアップロード時に圧縮パッケージから自動的にインポートされ、検証されます。

公式リポジトリにプラグインをアップロードする際に注意すべき検証ルールをいくつか紹介します：

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` があること
3. **メタデータテーブル** にリストされているすべての必須メタデータが存在すること
4. *version* メタデータフィールドがユニークであること
5. a license file must be included, saved as `LICENSE` with no extension (i.e. not `LICENSE.txt` for example)

プラグインの構造

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

```
plugin.zip
  pluginfolder/
  |-- i18n
  |   |-- translation_file_de.ts
  |-- img
  |   |-- icon.png
  |   |-- iconsource.svg
  |-- __init__.py
  |-- LICENSE
  |-- Makefile
```

(次のページに続く)

(前のページからの続き)

```
|-- metadata.txt
|-- more_code.py
|-- main_code.py
|-- README
|-- ui_Qt_user_interface_file.ui
```

Python プログラミング言語を使ってプラグインを作ることができます。 C++で書かれた従来のプラグインと比べると、Python 言語の動的な性質のおかげで、より書きやすく、より理解しやすく、そしてより配布しやすくなっています。

Python プラグインは、C++プラグインとともに、QGIS プラグインマネージャに表示さます。 これらのプラグインは、 ~/(UserProfile)/python/plugins と、以下のパスから検索されます。

- UNIX/Mac: (qgis_prefix)/share/qgis/python/plugins
- Windows: (qgis_prefix)/python/plugins

~ と (UserProfile) の定義は、core_and_external_plugins を参照してください。

注釈: *QGIS_PLUGINPATH* に既存のディレクトリ・パスを設定することにより、プラグインが検索されるパスのリストに、既存のパスを加えることができます。

第17章　プロセシングプラグインを書く

開発しようとしているプラ　　グインの種類によっては、プロセシングアルゴリズム（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フェースといった、「プロセシング」のコンポーネントの中で実行できるためです）、また開発時間の短縮も期待できます（「プロセシング」が作業の大部分を肩代わりしてくれるからです）。

開発したアルゴリズムを配布するためには、アルゴリズムをプロセシングツールボックスに追加するためのプラグインを新しく作る必要があります。このプラグインにはアルゴリズムプロバイダを含ませるとともに、プラグインの初期化の際にアルゴリズムがツールボックスに登録されるようにする必要があります。

17.1 イチから作る

アルゴリズムプロバイダを含むプラグインをイチから作るには、Plugin Builder を使って以下のステップに従います。

1. **Plugin Builder** プラグインをインストールする
2. Plugin Builder を使用して新しくプラグインを作成します。Plugin Builder が使用するテンプレートをきいてきたら、「プロセシングプロバイダ」を選択します。
3. 生成されたプラグインには、アルゴリズムをひとつ持つプロバイダが含まれています。プロバイダファイルおよびアルゴリズムファイルには両方ともに十分なコメントがついていて、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。

17.2 プラグインをアップデートする

すでに作成済みのプラグインをプロセッシングに追加したい場合は、さらにコードを追加する必要があります。

1. `metadata.txt` ファイルに以下の変数を追加する必要があります。

```
hasProcessingProvider=yes
```

2. `initGui` メソッドによってプラグインのセットアップを担う Python ファイルでは、幾つかのコードを直す必要があります。

```
from qgis.core import QgsApplication
from .processing_provider.provider import Provider

class YourPluginName:

    def __init__(self):
        self.provider = None

    def initProcessing(self):
        self.provider = Provider()
        QgsApplication.processingRegistry().addProvider(self.provider)

    def initGui(self):
        self.initProcessing()

    def unload(self):
        QgsApplication.processingRegistry().removeProvider(self.provider)
```

3. processing_provider フォルダを作ってそこに次の 3 つのファイルを納めることもできます。

   - 白紙の __init__.py ファイル。このファイルは妥当な Python パッケージを作るために必要です。
   - provider.py ファイルはプロセッシングプロバイダを生成しあなたのアルゴリズムを外部から使えるようにします。

```
from qgis.core import QgsProcessingProvider
from qgis.PyQt.QtGui import QIcon

from .example_processing_algorithm import ExampleProcessingAlgorithm


class Provider(QgsProcessingProvider):

    """ The provider of our plugin. """

    def loadAlgorithms(self):
        """ Load each algorithm into the current provider. """
        self.addAlgorithm(ExampleProcessingAlgorithm())
        # add additional algorithms here
        # self.addAlgorithm(MyOtherAlgorithm())

    def id(self) -> str:
        """The ID of your plugin, used for identifying the provider.

        This string should be a unique, short, character only string,
        eg "qgis" or "gdal". This string should not be localised.
        """
```

(次のページに続く)

(前のページからの続き)

```
        return 'yourplugin'

    def name(self) -> str:
        """The human friendly name of your plugin in Processing.

        This string should be as short as possible (e.g. "Lastools", not
        "Lastools version 1.0.1 64-bit") and localised.
        """
        return self.tr('Your plugin')

    def icon(self) -> QIcon:
        """Should return a QIcon which is used for your provider inside
        the Processing toolbox.
        """
        return QgsProcessingProvider.icon(self)
```

- example_processing_algorithm.py ファイルはサンプルアルゴリズムを含みます。 script template file の内容をコピー＆ペーストして、自分の必要に合わせて修正してください。

You should have a tree similar to this :

```
└─── your_plugin_root_folder
    ├─── __init__.py
    ├─── LICENSE
    ├─── metadata.txt
    └─── processing_provider
         ├─── example_processing_algorithm.py
         ├─── __init__.py
         └─── provider.py
```

1. ここまできたら QGIS でプラグインをリロードすれば、プロセッシングツールボックスとモデラーの中にあなたのスクリプトを見つけることができるはずです。

第18章　プラグインレイヤを使う

ヒント: pyqgis コンソールを使わない場合、このページにあるコードスニペットは次のインポートが必要です:

```
from qgis.core import (
    QgsPluginLayer,
    QgsPluginLayerType,
    QgsMapLayerRenderer,
    QgsApplication,
    QgsProject,
)

from qgis.PyQt.QtGui import QImage
```

もしプラグインがマップレイヤをレンダリングするために独自のメソッドを使うのであれば、QgsPluginLayer に基づいて独自のレイヤタイプを書くのが実装するための最良の方法かもしれません。

18.1 QgsPluginLayer のサブクラス化

以下は最小限の QgsPluginLayer の実装例です。これは Watermark example plugin のオリジナルのコードを基にしています。

カスタムレンダラーは、キャンバスへの実際の描画を定義する実装の一部です。

```
class WatermarkLayerRenderer(QgsMapLayerRenderer):

    def __init__(self, layerId, rendererContext):
        super().__init__(layerId, rendererContext)

    def render(self):
        image = QImage("/usr/share/icons/hicolor/128x128/apps/qgis.png")
        painter = self.renderContext().painter()
        painter.save()
        painter.drawImage(10, 10, image)
        painter.restore()
        return True
```

(次のページに続く)

(前のページからの続き)

```
class WatermarkPluginLayer(QgsPluginLayer):

    LAYER_TYPE="watermark"

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

    def createMapRenderer(self, rendererContext):
        return WatermarkLayerRenderer(self.id(), rendererContext)

    def setTransformContext(self, ct):
        pass

    # Methods for reading and writing specific information to the project file can
    # also be added:

    def readXml(self, node, context):
        pass

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

プラグインレイヤは、他のマップレイヤと同じようにプロジェクトやキャンバスに追加することができます:

```
plugin_layer = WatermarkPluginLayer()
QgsProject.instance().addMapLayer(plugin_layer)
```

このようなレイヤを含むプロジェクトをロードする場合、ファクトリークラスが必要です:

```
class WatermarkPluginLayerType(QgsPluginLayerType):

    def __init__(self):
        super().__init__(WatermarkPluginLayer.LAYER_TYPE)

    def createLayer(self):
        return WatermarkPluginLayer()

    # You can also add GUI code for displaying custom information
    # in the layer properties
    def showLayerProperties(self, layer):
        pass


```

(次のページに続く)

(前のページからの続き)

```
# Keep a reference to the instance in Python so it won't
# be garbage collected
plt =  WatermarkPluginLayerType()

assert QgsApplication.pluginLayerRegistry().addPluginLayerType(plt)
```

第19章　ネットワーク分析ライブラリ

ヒント: pyqgis コンソールを使わない場合、このページにあるコードスニペットは次のインポートが必要です:

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

ネットワーク分析ライブラリは次のことに使われます:

- 地理データ（ポリライン・ベクタレイヤ）から数学的グラフを作る
- グラフ理論の基本的な手法を実装する（現在はダイクストラ法のみ）

ネットワーク分析ライブラリは RoadGraph コアプラグインから基本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。

19.1 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的なグラフ（たいていはポリラインベクタレイヤ）を作成する
2. グラフ分析を実行する
3. 分析結果を利用する（例えば、その可視化）

19.2 グラフを構築する

最初にする必要がある事は --- 入力データを準備すること、つまりベクタレイヤ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タレイヤも使用できます。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 つの線は互いに接続しています。

さらに、グラフの作成時には、入力ベクタレイヤ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。

ベクタレイヤ属性と辺の長さは、辺のプロパティとして使用できます。

ベクタレイヤからグラフへの変換は、Builder プログラミングパターンを使って行われます。グラフはいわゆる Director を使って構築されます。今のところ Director は 1 つしかありません: `QgsVectorLayerDirector`。ディレクタは、ビルダがラインベクタレイヤからグラフを作るのに使う基本的な設定を行います。現在のところ、ディレクタと同様に、ビルダは 1 つしかありません: `QgsGraphBuilder` で、`QgsGraph` オブジェクトを作成します。BGL <https://www.boost.org/doc/libs/1_48_0/libs/graph/doc/index.html>`_ や NetworkX などのライブラリと互換性のあるグラフを構築する独自のビルダを実装することもできます。

辺のプロパティを計算するために、プログラミングパターン strategy が使用されます。今のところ（経路の長さを考慮する）`QgsNetworkDistanceStrategy` 戦略と（速度も考慮する）:class:*QgsNetworkSpeedStrategy <qgis.analysis.QgsNetworkSpeedStrategy>* 戦略のみが利用可能です。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することができます。例えば、RoadGraph プラグインでは、属性から辺の長さと速度の値を使用して移動時間を計算する戦略を使用しています。

では手順に行きましょう。

まず、このライブラリを使うには、解析モジュールをインポートする必要があります

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

それからディレクタを作成するためのいくつかの例

```
# don't use information about road direction from layer attributes,
# all roads are treated as two-way
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '', QgsVectorLayerDirector.
↪DirectionBoth)

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

ディレクタを構築するには、グラフ構造のソースとなるベクタレイヤと、各道路セグメントで許容される移動に関する情報（一方通行か双方向移動か、直進か逆走か）を渡す必要があります。呼び出しは次のようになります

```
director = QgsVectorLayerDirector(vectorLayer,
                                  directionFieldId,
                                  directDirectionValue,
                                  reverseDirectionValue,
                                  bothDirectionValue,
```

(次のページに続く)

(前のページからの続き)

```
6                                                   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：

- vectorLayer --- グラフを作るのに使われるベクタレイヤ
- 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 ならこの情報をまったく使用されません。整数。
- directDirectionValue ---順方向の道路（線の最初の点から最後の点へ移動）のフィールド値。文字列。
- reverseDirectionValue ---逆方向の道路（線の最後の点から最初の点へ移動）のフィールド値。文字列。
- bothDirectionValue ---双方向道路（最初の点から最後の点まで、且つ、最後の点から最初の点まで移動できるような道路）のフィールド値。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、または上記の 3 つの値のいずれとも異なる値を持つ道路に使用されます。設定可能な値は以下の通りです：
  - QgsVectorLayerDirector.DirectionForward --- 一方向順方向
  - QgsVectorLayerDirector.DirectionBackward --- 一方向逆方向
  - QgsVectorLayerDirector.DirectionBoth --- 両方向

それから、辺のプロパティを計算するための戦略を作成することが必要です

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

そして、ディレクタにこの戦略について教えます

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

これでグラフを作成するビルダを使うことができます。QgsGraphBuilder クラスのコンストラクタはいくつかの引数を取ります：

- crs --- 使用する座標参照系。必須引数。
- otfEnabled --- 「オンザフライ」再投影を使用するかどうか。デフォルトは True （OTF を使う）。
- topologyTolerance --- トポロジの許容範囲。デフォルト値は 0。
- ellipsoidID --- 使用する楕円体。デフォルトは "WGS84"。

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

分析に使用されるいくつかの点を定義することもできます。例えば

```
startPoint = QgsPointXY(1179720.1871, 5419067.3507)
endPoint = QgsPointXY(1180616.0205, 5419745.7839)
```

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（レイヤ中の地物数とレイヤの大きさに応じて）いくらか時間がかかることがあります。 tiedPoints は「結びつけられた」点の座標を持つリストです。ビルド操作が完了するとグラフが得られ、それを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])
```

19.3 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ーの値を見つけます。結果は、最短経路木として表現できます。

最短経路木は、以下の性質を持つ有向加重グラフ（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 クラスの shortestTree() と dijkstra() メソッドを使用します。dijkstra() メソッドを使用することを推奨します。なぜなら、このメソッドはより速く動作し、より効率的にメモリを使用するからです。

shortestTree() メソッドは最短経路木を様々な角度から考えたいときに便利です。それは常に新しいグラフオブジェクト (QgsGraph) を生成し、3 つの変数を受け取ります：

- source --- 入力グラフ

- startVertexIdx --- 木上の点のインデックス（木の根）
- criterionNum --- 使用する辺のプロパティの数（0 から始まる）。

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

dijkstra() メソッドは同じ引数を持ちますが、2 つの配列を返します。最初の配列要素 *n* には入力辺のインデックス、または、入力辺がないときは-1 が格納されます。2 番目の配列の要素 *n* には、木の根から頂点 *n* までの距離、または頂点 *n* が根から到達できない場合は DOUBLE_MAX が格納されます。

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

以下は、shortestTree() メソッドで作成したグラフを使用して、最短経路木を表示する非常に簡単なコードです（ **レイヤ** パネルでラインストリングレイヤを選択し、座標をあなたのものに置き換えてください）。

警告: このコードは例としてだけ使ってください。これは QgsRubberBand オブジェクトを大量に生成するので、大きなデータセットでは遅くなるかもしれません。

```
from qgis.core import *
from qgis.gui import *
from qgis.analysis import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *

vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines', 'lines')
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '', QgsVectorLayerDirector.
↪DirectionBoth)
strategy = QgsNetworkDistanceStrategy()
director.addStrategy(strategy)
builder = QgsGraphBuilder(vectorLayer.crs())

pStart = QgsPointXY(1179661.925139,5419188.074362)
tiedPoint = director.makeGraph(builder, [pStart])
pStart = tiedPoint[0]

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

i = 0
while (i < tree.edgeCount()):
  rb = QgsRubberBand(iface.mapCanvas())
  rb.setColor (Qt.red)
```

(次のページに続く)

(前のページからの続き)

```
  rb.addPoint (tree.vertex(tree.edge(i).fromVertex()).point())
  rb.addPoint (tree.vertex(tree.edge(i).toVertex()).point())
  i = i + 1
```

同じことですが、`dijkstra()` メソッドを使います

```
from qgis.core import *
from qgis.gui import *
from qgis.analysis import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *

vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines', 'lines')

director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '', QgsVectorLayerDirector.
↪DirectionBoth)
strategy = QgsNetworkDistanceStrategy()
director.addStrategy(strategy)
builder = QgsGraphBuilder(vectorLayer.crs())

pStart = QgsPointXY(1179661.925139,5419188.074362)
tiedPoint = director.makeGraph(builder, [pStart])
pStart = tiedPoint[0]

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

for edgeId in tree:
  if edgeId == -1:
    continue
  rb = QgsRubberBand(iface.mapCanvas())
  rb.setColor (Qt.red)
  rb.addPoint (graph.vertex(graph.edge(edgeId).fromVertex()).point())
  rb.addPoint (graph.vertex(graph.edge(edgeId).toVertex()).point())
```

19.3.1 最短経路を見つける

2 点間の最適経路を求めるには、次のようなアプローチが用いられます。両方の点（始点 A と終点 B）はグラフが構築されるときそれに「結び付け」されています。その後、 `shortestTree()` または `dijkstra()` メソッドを用いて、始点 A を根とする最短経路木を構築します。 同じ木で終点 B も求め、点 B から点 A まで木を歩き始めます。アルゴリズム全体は次のように書けます：

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

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。

以下は、QGIS Python コンソールで `shortestTree()` メソッドを使うサンプルコードです（TOC にあるラインストリングレイヤを読み込んで選択し、コード内の座標を自分のものに置き換える必要があるかもしれません）

```
from qgis.core import *
from qgis.gui import *
from qgis.analysis import *

from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *

vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines', 'lines')
builder = QgsGraphBuilder(vectorLayer.sourceCrs())
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '', QgsVectorLayerDirector.
↪DirectionBoth)
strategy = QgsNetworkDistanceStrategy()
director.addStrategy(strategy)

startPoint = QgsPointXY(1179661.925139,5419188.074362)
endPoint = QgsPointXY(1180942.970617,5420040.097560)

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

graph = builder.graph()
idxStart = graph.findVertex(tStart)

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

```

(次のページに続く)

(前のページからの続き)

```
idxStart = tree.findVertex(tStart)
idxEnd = tree.findVertex(tStop)

if idxEnd == -1:
    raise Exception('No route!')

# Add last point
route = [tree.vertex(idxEnd).point()]

# Iterate the graph
while idxEnd != idxStart:
    edgeIds = tree.vertex(idxEnd).incomingEdges()
    if len(edgeIds) == 0:
        break
    edge = tree.edge(edgeIds[0])
    route.insert(0, tree.vertex(edge.fromVertex()).point())
    idxEnd = edge.fromVertex()

# Display
rb = QgsRubberBand(iface.mapCanvas())
rb.setColor(Qt.green)

# This may require coordinate transformation if project's CRS
# is different than layer's CRS
for p in route:
    rb.addPoint(p)
```

以下は同じサンプルですが、 `dijkstra()` メソッドを使用しています

```
from qgis.core import *
from qgis.gui import *
from qgis.analysis import *

from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *

vectorLayer = QgsVectorLayer('testdata/network.gpkg|layername=network_lines', 'lines')
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '', QgsVectorLayerDirector.
↪DirectionBoth)
strategy = QgsNetworkDistanceStrategy()
director.addStrategy(strategy)

builder = QgsGraphBuilder(vectorLayer.sourceCrs())

startPoint = QgsPointXY(1179661.925139,5419188.074362)
```

(次のページに続く)

(前のページからの続き)

```
endPoint = QgsPointXY(1180942.970617,5420040.097560)

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

graph = builder.graph()
idxStart = graph.findVertex(tStart)
idxEnd = graph.findVertex(tStop)

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

if tree[idxEnd] == -1:
    raise Exception('No route!')

# Total cost
cost = costs[idxEnd]

# Add last point
route = [graph.vertex(idxEnd).point()]

# Iterate the graph
while idxEnd != idxStart:
    idxEnd = graph.edge(tree[idxEnd]).fromVertex()
    route.insert(0, graph.vertex(idxEnd).point())

# Display
rb = QgsRubberBand(iface.mapCanvas())
rb.setColor(Qt.red)

# This may require coordinate transformation if project's CRS
# is different than layer's CRS
for p in route:
    rb.addPoint(p)
```

19.3.2 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ます。「消防署があります。消防車が 5 分、10 分、15 分で到達できるのは市内のどの部分ですか？」。これらの質問への回答が消防署の利用可能領域です。

利用可能領域を見つけるには、`QgsGraphAnalyzer` クラスの `dijkstra()` メソッドを使用します。cost 配列の要素をあらかじめ定義された値と比較するだけです。もし cost[i] があらかじめ定義された値以下であれば、頂点 i は利用可能領域内であり、そうでなければ領域外です。

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、辺の元頂点が到達可能であり辺の先頂点が到達可能ではないような、最短経路木の辺に基づく利用可能境界です。

例です

```
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '', QgsVectorLayerDirector.
↪DirectionBoth)
strategy = QgsNetworkDistanceStrategy()
director.addStrategy(strategy)
builder = QgsGraphBuilder(vectorLayer.crs())


pStart = QgsPointXY(1179661.925139, 5419188.074362)
delta = iface.mapCanvas().getCoordinateTransform().mapUnitsPerPixel() * 1

rb = QgsRubberBand(iface.mapCanvas())
rb.setColor(Qt.green)
rb.addPoint(QgsPointXY(pStart.x() - delta, pStart.y() - delta))
rb.addPoint(QgsPointXY(pStart.x() + delta, pStart.y() - delta))
rb.addPoint(QgsPointXY(pStart.x() + delta, pStart.y() + delta))
rb.addPoint(QgsPointXY(pStart.x() - delta, pStart.y() + delta))

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

idStart = graph.findVertex(tStart)

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

upperBound = []
r = 1500.0
i = 0
tree.reverse()

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

for i in upperBound:
    centerPoint = graph.vertex(i).point()
    rb = QgsRubberBand(iface.mapCanvas())
    rb.setColor(Qt.red)
```

(次のページに続く)

(前のページからの続き)

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

第20章　QGIS ServerとPython

20.1 はじめに

QGIS サーバーについて更に学ぶには、 QGIS-Server-manual を読んでください。

QGIS Server は 3 つの異なるものです:

1. QGIS Server ライブラリ：OGC ウェブサービスを作るための API を提供するライブラリ
2. QGIS Server FCGI: ウェブサーバーと共に OGC サービス（WMS、WFS、WCS など）と OGC API (WFS3/OAPIF) を実装する FCGI バイナリアプリケーション `qgis_mapserv.fcgi`
3. QGIS 開発サーバー: OGC サービス（WMS、WFS、WCS など）と OGC API (WFS3/OAPIF) を実装した開発サーバーバイナリーアプリケーション `qgis_mapserver`

クックブックのこの章では、最初のトピックに焦点を当て、QGIS Server API の使用方法を説明することで、Python を使ってサーバーの動作を拡張、強化、カスタマイズする方法や、QGIS Server API を使って QGIS サーバーを別のアプリケーションに組み込む方法を示します。

あなたが直面する可能性のある主なシナリオである、QGIS Server の動作を変更したり、機能を拡張して新しいカスタムサービスや API を提供したりするには、いくつかの方法があります:

- EMBEDDING → 別の Python アプリケーションから QGIS Server API を使用します
- STANDALONE → QGIS Server をスタンドアロンの WSGI/HTTP サービスとして実行します
- FILTERS → フィルタープラグインで QGIS サーバーを強化/カスタマイズします
- SERVICES → 新しい *SERVICE* を追加します
- OGC APIs → 新しい *OGC API* を追加します

埋め込みアプリケーションとスタンドアロンアプリケーションでは、他の Python スクリプトやアプリケーションから QGIS Server Python API を直接使用する必要があります。残りのオプションでは、標準の QGIS Server バイナリアプリケーション（FCGI または開発サーバー）にカスタム機能を追加する場合に適しています: この場合、サーバーアプリケーション用の Python プラグインを書いてカスタムフィルタ、サービス、または API を登録する必要があります。

20.2 Server API の基本

典型的な QGIS Server アプリケーションの基本クラスは以下の通りです:

- `QgsServer` サーバーのインスタンス（通常はアプリケーション全体で一つのインスタンス）
- `QgsServerRequest` リクエストオブジェクト（通常はリクエスト毎に再生成されます。）
- `QgsServer.handleRequest(request, response)` はリクエストを処理し、レスポンスを生成します

QGIS Server FCGI または開発サーバーのワークフローをまとめると、以下のようになります:

```
initialize the QgsApplication
create the QgsServer
the main server loop waits forever for client requests:
    for each incoming request:
        create a QgsServerRequest request
        create a QgsServerResponse response
        call QgsServer.handleRequest(request, response)
            filter plugins may be executed
        send the output to the client
```

`QgsServer.handleRequest(request, response)` メソッドの内部では、フィルタプラグインのコールバックが呼び出され、`QgsServerRequest` と `QgsServerResponse` は `QgsServerInterface` クラスを通してプラグインから利用できるようになります。

警告: QGIS サーバークラスはスレッドセーフではないため、QGIS Server API に基づいたスケーラブルなアプリケーションを構築する場合は、常にマルチプロセッシングモデルまたはコンテナを使用する必要があります。

20.3 STANDALONE 又は EMBEDDING

スタンドアロン・サーバー・アプリケーションや組み込みの場合は、上記のサーバー・クラスを直接使用し、クライアントとのすべての HTTP プロトコルのやり取りを管理するウェブ・サーバー実装にまとめる必要があります。

QGIS Server API の最小限の使用例（HTTP 部分を除く）を以下に示します:

```
from qgis.core import QgsApplication
from qgis.server import *
app = QgsApplication([], False)

# Create the server instance, it may be a single one that
# is reused on multiple requests
server = QgsServer()

```

(次のページに続く)

(前のページからの続き)

```
# Create the request by specifying the full URL and an optional body
# (for example for POST requests)
request = QgsBufferServerRequest(
    'http://localhost:8081/?MAP=/qgis-server/projects/helloworld.qgs' +
    '&SERVICE=WMS&REQUEST=GetCapabilities')

# Create a response objects
response = QgsBufferServerResponse()

# Handle the request
server.handleRequest(request, response)

print(response.headers())
print(response.body().data().decode('utf8'))

app.exitQgis()
```

これは、QGIS ソースコードリポジトリの継続的な統合テストのために開発された完全なスタンドアロンアプリケーションの例で、さまざまなプラグインフィルタと認証スキームが紹介されています（本番用ではなく、テスト目的のみで開発されていますが、学習には興味深いものです）：qgis_wrapped_server.py

20.4 サーバー・プラグイン

Server python プラグインは、QGIS Server アプリケーションの起動時に一度だけロードされ、フィルタ、サービス、または API を登録するために使用できます。

サーバープラグインの構造はデスクトップとよく似ており、`QgsServerInterface` オブジェクトがプラグインに提供され、プラグインはサーバーインターフェースによって公開されるメソッドの 1 つを使用して、対応するレジストリに 1 つ以上のカスタムフィルタ、サービス、または API を登録することができます。

20.4.1 サーバー・フィルター・プラグイン

フィルタには 3 つの異なるフレーバーがあり、以下のクラスの 1 つをサブクラス化し、`QgsServerInterface` の対応するメソッドを呼び出すことでインスタンス化できます:

フィルタ型	ベースクラス	QgsServerInterface 登録
I/O	`QgsServerFilter`	`registerFilter()`
アクセス制御	`QgsAccessControlFilter`	`registerAccessControl()`
キャッシュ	`QgsServerCacheFilter`	`registerServerCache()`

i/O フィルタ

I/O フィルタは、コアサービス（WMS、WFS など）のサーバー入力と出力（リクエストとレスポンス）を変更し、サービスのワークフローにあらゆる操作を加えることができます。例えば、選択したレイヤーへのアクセスを制限したり、XML レスポンスに XSL スタイルシートを注入したり、生成された WMS 画像に透かしを追加したりすることができます。

この時点で、 server plugins API docs をざっと見ておくと役に立つかもしれません。

各フィルタは、3 つのコールバックのうち少なくとも 1 つを実装する必要があります:

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

全てのフィルタはリクエスト／レスポンスオブジェクト (`QgsRequestHandler`) にアクセスすることができ、その全てのプロパティ (入力/出力) を操作し、例外を発生させることができます（後述するように、かなり特殊な方法で)。

これらのメソッドはすべて、その呼び出しが後続のフィルタに伝搬されるべきかどうかを示すブール値を返します。これらのメソッドのいずれかが `False` を返した場合はチェインが停止し、そうでない場合は呼び出しが次のフィルタに伝搬します。

以下は、典型的なリクエストをサーバがどのように処理し、フィルタのコールバックがいつ呼ばれるかを示す擬似コードです:

```
for each incoming request:
    create GET/POST request handler
    pass request to an instance of QgsServerInterface
    call onRequestReady filters

    if there is not a response:
        if SERVICE is WMS/WFS/WCS:
            create WMS/WFS/WCS service
            call service's executeRequest
                possibly call onSendResponse for each chunk of bytes
                sent to the client by a streaming services (WFS)
        call onResponseComplete
    request handler sends the response to the client
```

次の段落では、利用可能なコールバックを詳細に説明します。

onRequestReady

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、これは入力を操作するなどのアクションを実行できるポイントです。

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

SERVICE パラメーターを変更することでコアサービスを完全に置き換え、それによりコアサービスを完全にバイパスすることさえできるかもしれません（とはいえ、これはあまり意味がないということ）。

onSendResponse

これは、部分的な出力がレスポンスバッファから (つまり fcgi サーバが使用されている場合は **FCGI** `stdout` に)、さらにそこからクライアントにフラッシュされるたびに呼び出されます。これは（WFS の GetFeature のように）巨大なコンテンツがストリームされる場合に発生します。この場合、 `onSendResponse()` が複数回呼び出される可能性があります。

レスポンスがストリームされない場合、 `onSendResponse()` は全く呼ばれないことに注意してください。

全ての場合において、最後の (あるいはユニークな) チャンクは `onResponseComplete()` の呼び出しの後にクライアントに送られます。

`False` を返すと、クライアントへのデータのフラッシュを防ぎます。プラグインがレスポンスから全てのチャンクを収集し、 `onResponseComplete()` でレスポンスの検査や変更を行いたい場合、これは望ましいことです。

onResponseComplete

これは、コアサービス（ヒットした場合）が処理を終了し、リクエスト をクライアントに送る準備ができたときに一度だけ呼ばれます。上述したように、このメソッドは最後の（あるいはユニークな）データチャンクがクライアントに送信される前に呼ばれます。ストリーミングサービスの場合、 `onSendResponse()` への複数の呼び出しが呼び出されている可能性があります。

`onResponseComplete()` は、新しいサービスの実装（WPS やカスタムサービス）を提供したり、コアサービスからの出力を直接操作する（例えば、WMS 画像に透かしを追加する）のに最適な場所です。

`False` を返すと、次のプラグインが `onResponseComplete()` を実行できなくなりますが、いずれにせよ、レスポンスがクライアントに送信されなくなることに注意してください。

プラグインから例外を発生させる

このトピックについては、まだいくつかの仕事が残っています: 現在の実装では、QgsMapServiceExceptionのインスタンスに `QgsRequestHandler` プロパティを設定することで、処理される例外と処理されない例外を区別することができます。こうすることで、メインの C++コードは、処理された python の例外をキャッチし、処理されなかった例外を無視（またはそれを記録）することができます。

このアプローチは、基本的に動作しますが、それは非常に「パイソン的」ではありません：より良いアプローチは、Python コードから例外を発生し、それらがそこで処理されるために C ++ループに湧き上がるのを見ることでしょう。

サーバー・プラグインを書く

サーバプラグインは *Python* **プラグインを開発する** で説明されているような標準的な QGIS Python プラグインであり、追加の（あるいは代替の）インターフェースを提供します：一般的な QGIS デスクトッププラグインは `QgisInterface` のインスタンスを通して QGIS アプリケーションにアクセスすることができますが、サーバプラグインは QGIS Server アプリケーションコンテキスト内で実行された時のみ `QgsServerInterface` にアクセスすることができます。

プラグインがサーバーインターフェイスを持っていることを QGIS Server に認識させるには、特別なメタデータエントリが（`metadata.txt` に）必要です:

```
server=True
```

重要: `server=True` のメタデータが設定されているプラグインだけが QGIS Server に読み込まれ、実行されます。

ここで説明した qgis3-server-vagrant のサンプルプラグインは github で公開されています（他にもたくさんあります）。いくつかのサーバプラグインは公式の QGIS plugins リポジトリ でも公開されています。

プラグインファイル

以下は、サーバープラグイン例のディレクトリ構造です。

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

__init__.py

このファイルは Python のインポートシステムによって要求されます。また、QGIS Server はこのファイルに serverClassFactory() 関数を含めることを要求します。この関数は、サーバーが開始して、QGIS Server にプラグインが読み込まれるときに呼び出されます。それは QgsServerInterface のインスタンスへの参照を受け取り、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。プラグインの例 __init__.py はこのようになっています:

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

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 HelloServer.py)

サーバプラグインは通常 1 つ以上のコールバックで構成され、 QgsServerFilter のインスタンスにパックされます。

各 QgsServerFilter は以下のコールバックのひとつ以上を実装しています:

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

次の例は、**SERVICE** パラメーターが "HELLO "に等しい場合に、*HelloServer!* を印字する最小のフィルタを実施しています:

```
class HelloFilter(QgsServerFilter):

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

    def onRequestReady(self) -> bool:
        QgsMessageLog.logMessage("HelloFilter.onRequestReady")
        return True

    def onSendResponse(self) -> bool:
        QgsMessageLog.logMessage("HelloFilter.onSendResponse")
        return True

    def onResponseComplete(self) -> bool:
        QgsMessageLog.logMessage("HelloFilter.onResponseComplete")
        request = self.serverInterface().requestHandler()
        params = request.parameterMap()
        if params.get('SERVICE', '').upper() == 'HELLO':
```

(次のページに続く)

(前のページからの続き)

```
            request.clear()
            request.setResponseHeader('Content-type', 'text/plain')
            # Note that the content is of type "bytes"
            request.appendBody(b'HelloServer!')
        return True
```

フィルターは、以下の例のように、**serverIface** に登録されなければなりません:

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

`registerFilter())` の 2 番目のパラメータは、同じ名前のコールバックの順番を定義する優先度を設定します（優先度の低いものが最初に呼び出されます）。

3 つのコールバックを使うことで、プラグインは様々な方法でサーバの入力や出力を操作することができます。プラグインのインスタンスはいつでも `QgsServerInterface` を通して `QgsRequestHandler` にアクセスすることができます。`QgsRequestHandler` クラスには、サーバのコア処理に入る前（:func:`requestReady`を使用する）や、リクエストがコアサービスで処理された後（:func:`sendResponse`を使用する）に入力パラメータを変更するために使用できるメソッドがたくさんあります。

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：

入力を変更する

プラグイン例には、クエリ文字列から来る入力パラメータを変えるテスト例が含まれています。この例では、新しいパラメータが（すでにパースされた）``parameterMap`` に注入され、このパラメータはコアサービス（WMS など）から見えるようになります：

```
class ParamsFilter(QgsServerFilter):

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

    def onRequestReady(self) -> bool:
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        request.setParameter('TEST_NEW_PARAM', 'ParamsFilter')
        return True

    def onResponseComplete(self) -> bool:
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        if params.get('TEST_NEW_PARAM') == 'ParamsFilter':
            QgsMessageLog.logMessage("SUCCESS - ParamsFilter.onResponseComplete")
        else:
```

(次のページに続く)

(前のページからの続き)

```
            QgsMessageLog.logMessage("FAIL    - ParamsFilter.onResponseComplete")
        return True
```

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

強調表示された行の「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。

同じテクニックを使えば、コアサービスの代わりにカスタムサービスを使うこともできます: 例えば、**WFS SERVICE** リクエストやその他のコアリクエストを、**SERVICE** パラメータを別のものに変更するだけで、コアサービスをスキップすることができます。そして、カスタム結果を出力に注入してクライアントに送ることができます（これについては後述します）。

Tip: 本当にカスタムサービスを実装したい場合は、`QgsService` をサブクラスにして、`registerFilter()` の `registerService(service)` を呼び出してサービスを登録することをお勧めします。

出力を変更または置き換えする

透かしフィルタの例は、WMS コアサービスによって作成された WMS 画像の上に透かし画像を加算した新たな画像で WMS 出力を置き換える方法を示しています：

```
from qgis.server import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *

class WatermarkFilter(QgsServerFilter):

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

    def onResponseComplete(self) -> bool:
        request = self.serverInterface().requestHandler()
```

(次のページに続く)

(前のページからの続き)

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

この例では、**SERVICE** パラメータ値がチェックされ、受信リクエストが **WMS GETMAP** であり、以前に実行されたプラグインまたはコアサービス（この場合は WMS）によって例外が設定されていない場合、WMS によって生成された画像が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、新しく生成された画像に置き換えることです。実際の状況では、PNG や JPG だけをサポートするのではなく、要求された画像タイプもチェックする必要があることに注意してください。

アクセス制御フィルタ

アクセス制御フィルタは、開発者がどのレイヤー、機能、属性にアクセスできるかを細かく制御できるようにします。アクセス制御フィルタには、以下のコールバックを実装できます：

- `layerFilterExpression(layer)`
- `layerFilterSubsetString(layer)`
- `layerPermissions(layer)`
- `authorizedLayerAttributes(layer, attributes)`
- `allowToEdit(layer, feature)`
- `cacheKey()`

プラグインファイル

以下は、プラグイン例のディレクトリ構造です:

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

__init__.py

このファイルは Python のインポートシステムによって必要とされます。全ての QGIS サーバプラグインと同様に、このファイルには `serverClassFactory()` 関数が含まれており、プラグインが起動時に QGIS Server にロードされる際に呼び出されます。この関数は `QgsServerInterface` のインスタンスへの参照を受け取り、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。プラグインの例 `__init__.py` はこのようになります：

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

AccessControl.py

```
class AccessControlFilter(QgsAccessControlFilter):

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

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

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

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

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

(次のページに続く)

(前のページからの続き)

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

    def cacheKey(self):
        return super().cacheKey()

class AccessControlServer:

   def __init__(self, serverIface):
      """ Register AccessControlFilter """
      serverIface.registerAccessControl(AccessControlFilter(serverIface), 100)
```

この例では全員に完全なアクセス権を与えています。

誰がログオンしているかを知るのはこのプラグインの役割です。

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

layerFilterExpression

結果を制限する式を追加するために使用されます。

例えば、属性 role が user に等しい地物に制限します。

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

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

例えば、属性 role が user に等しい地物に制限します。

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

layerPermissions

レイヤーへのアクセスを制限します。

プロパティを持つ `LayerPermissions()` 型のオブジェクトを返します：

- `canRead` で `GetCapabilities` に表示され、読み取りアクセスが可能になります。
- `canInsert` で新しい地物を挿入できるようになります。
- `canUpdate` で地物を更新できるようになります。
- `canDelete` で地物を削除できるようになります。

例えば、すべてを読み取り専用アクセスに制限するには：

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

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。

例えば、`role` 属性を隠します:

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

allowToEdit

これは、地物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。

例えば、`role` という属性に `user` という値を持つ地物だけを編集できるようにします：

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

cacheKey

QGIS Server は機能のキャッシュを維持し、role ごとにキャッシュを持つには、このメソッドで role を返します。また、キャッシュを完全に無効にするには None を返します。

20.4.2 カスタムサービス

QGIS Server では、WMS、WFS、WCS などのコアサービスは QgsService のサブクラスとして実装されています。

クエリ文字列パラメータ SERVICE がサービス名にマッチした時に実行される新しいサービスを実装するには、独自の QgsService を実装し、その registerService(service) を呼び出して serviceRegistry() にサービスを登録します。

以下は CUSTOM という名前のカスタムサービスの例です:

```
from qgis.server import QgsService
from qgis.core import QgsMessageLog

class CustomServiceService(QgsService):

    def __init__(self):
        QgsService.__init__(self)

    def name(self):
        return "CUSTOM"

    def version(self):
        return "1.0.0"

    def executeRequest(self, request, response, project):
        response.setStatusCode(200)
        QgsMessageLog.logMessage('Custom service executeRequest')
        response.write("Custom service executeRequest")


class CustomService():

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

20.4.3 カスタム API

QGIS Server では、OAPIF（別名 WFS3）のようなコア OGC API は QgsServerOgcApiHandler サブクラスのコレクションとして実装されており、QgsServerOgcApi (またはその親クラス QgsServerApi) のインスタンスに登録されます。

url パスが特定の URL にマッチした時に実行される新しい API を実装するには、独自の QgsServerOgcApiHandler インスタンスを実装し、それらを QgsServerOgcApi の registerApi(api) を呼び出して API を登録します。

以下は、URL に /customapi が含まれる場合に実行されるカスタム API の例です:

```
import json
import os

from qgis.PyQt.QtCore import QBuffer, QIODevice, QTextStream, QRegularExpression
from qgis.server import (
    QgsServiceRegistry,
    QgsService,
    QgsServerFilter,
    QgsServerOgcApi,
    QgsServerQueryStringParameter,
    QgsServerOgcApiHandler,
)

from qgis.core import (
    QgsMessageLog,
    QgsJsonExporter,
    QgsCircle,
    QgsFeature,
    QgsPoint,
    QgsGeometry,
)


class CustomApiHandler(QgsServerOgcApiHandler):

    def __init__(self):
        super(CustomApiHandler, self).__init__()
        self.setContentTypes([QgsServerOgcApi.HTML, QgsServerOgcApi.JSON])

    def path(self):
        return QRegularExpression("/customapi")

    def operationId(self):
        return "CustomApiXYCircle"

```

(次のページに続く)

(前のページからの続き)

```
    def summary(self):
        return "Creates a circle around a point"

    def description(self):
        return "Creates a circle around a point"

    def linkTitle(self):
        return "Custom Api XY Circle"

    def linkType(self):
        return QgsServerOgcApi.data

    def handleRequest(self, context):
        """Simple Circle"""

        values = self.values(context)
        x = values['x']
        y = values['y']
        r = values['r']
        f = QgsFeature()
        f.setAttributes([x, y, r])
        f.setGeometry(QgsCircle(QgsPoint(x, y), r).toCircularString())
        exporter = QgsJsonExporter()
        self.write(json.loads(exporter.exportFeature(f)), context)

    def templatePath(self, context):
        # The template path is used to serve HTML content
        return os.path.join(os.path.dirname(__file__), 'circle.html')

    def parameters(self, context):
        return [QgsServerQueryStringParameter('x', True,␣
↪QgsServerQueryStringParameter.Type.Double, 'X coordinate'),
                QgsServerQueryStringParameter(
                    'y', True, QgsServerQueryStringParameter.Type.Double, 'Y␣
↪coordinate'),
                QgsServerQueryStringParameter('r', True,␣
↪QgsServerQueryStringParameter.Type.Double, 'radius')]


class CustomApi():

    def __init__(self, serverIface):
        api = QgsServerOgcApi(serverIface, '/customapi',
                              'custom api', 'a custom api', '1.1')
        handler = CustomApiHandler()
```

(次のページに続く)

(前のページからの続き)

```
78          api.registerHandler(handler)
79          serverIface.serviceRegistry().registerApi(api)
```

第21章　PyQGISチートシート

ヒント: pyqgis コンソールを使わない場合、このページにあるコードスニペットは次のインポートが必要です:

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

from qgis.core import (
    Qgis,
    QgsProject,
    QgsLayerTreeModel,
)

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

21.1 ユーザーインターフェース

ルックアンドフィールの変更

```
from qgis.PyQt.QtWidgets import QApplication

app = QApplication.instance()
app.setStyleSheet(".QWidget {color: blue; background-color: yellow;}")
# You can even read the stylesheet from a file
with  open("testdata/file.qss") as qss_file_content:
    app.setStyleSheet(qss_file_content.read())
```

アイコンとタイトルの変更

```
from qgis.PyQt.QtGui import QIcon

icon = QIcon("/path/to/logo/file.png")
```

(次のページに続く)

(前のページからの続き)

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

21.2 設定

QgsSettings リストを取得する

```
from qgis.core import QgsSettings

qs = QgsSettings()

for k in sorted(qs.allKeys()):
    print (k)
```

21.3 ツールバー

ツールバーを削除する

```
toolbar = iface.helpToolBar()
parent = toolbar.parentWidget()
parent.removeToolBar(toolbar)

# and add again
parent.addToolBar(toolbar)
```

アクションツールバーを削除する

```
actions = iface.attributesToolBar().actions()
iface.attributesToolBar().clear()
iface.attributesToolBar().addAction(actions[4])
iface.attributesToolBar().addAction(actions[3])
```

21.4 メニュー

メニューを削除する

```
# for example Help Menu
menu = iface.helpMenu()
menubar = menu.parentWidget()
menubar.removeAction(menu.menuAction())
```

(次のページに続く)

(前のページからの続き)

```
5
6 # and add again
7 menubar.addAction(menu.menuAction())
```

21.5 キャンバス

キャンバスにアクセスする

```
canvas = iface.mapCanvas()
```

キャンバスの色を変更する

```
from qgis.PyQt.QtCore import Qt

iface.mapCanvas().setCanvasColor(Qt.black)
iface.mapCanvas().refresh()
```

マップの更新間隔

```
from qgis.core import QgsSettings
# Set milliseconds (150 milliseconds)
QgsSettings().setValue("/qgis/map_update_interval", 150)
```

21.6 レイヤー

ベクターレイヤーを追加する

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

アクティブなレイヤーを取得する

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

レイヤーをすべて一覧する

```
from qgis.core import QgsProject

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

レイヤー名を取得する

```
from qgis.core import QgsVectorLayer
layer = QgsVectorLayer("Point?crs=EPSG:4326", "layer name you like", "memory")
QgsProject.instance().addMapLayer(layer)

layers_names = []
for layer in QgsProject.instance().mapLayers().values():
    layers_names.append(layer.name())

print("layers TOC = {}".format(layers_names))
```

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

もしくは

```
layers_names = [layer.name() for layer in QgsProject.instance().mapLayers().values()]
print("layers TOC = {}".format(layers_names))
```

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

名前からレイヤーを見つける

```
from qgis.core import QgsProject

layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
print(layer.name())
```

```
layer name you like
```

アクティブなレイヤーを設定する

```
from qgis.core import QgsProject

layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
iface.setActiveLayer(layer)
```

間隔をおいてレイヤーを更新する

```
from qgis.core import QgsProject

layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
# Set seconds (5 seconds)
layer.setAutoRefreshInterval(5000)
# Enable data reloading
layer.setAutoRefreshMode(Qgis.AutoRefreshMode.ReloadData)
```

メソッドを表示する

```
dir(layer)
```

地物フォームを使って新たに地物を追加する

```
from qgis.core import QgsFeature, QgsGeometry

feat = QgsFeature()
geom = QgsGeometry()
feat.setGeometry(geom)
feat.setFields(layer.fields())

iface.openFeatureForm(layer, feat, False)
```

地物フォームを使わずに新たに地物を追加する

```
from qgis.core import QgsGeometry, QgsPointXY, QgsFeature

pr = layer.dataProvider()
feat = QgsFeature()
feat.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(10,10)))
pr.addFeatures([feat])
```

地物を取得する

```
for f in layer.getFeatures():
    print (f)
```

```
<qgis._core.QgsFeature object at 0x7f45cc64b678>
```

選択された地物を取得する

```
for f in layer.selectedFeatures():
    print (f)
```

選択された地物の ID を取得する

```
selected_ids = layer.selectedFeatureIds()
print(selected_ids)
```

選択された地物の ID からメモリに一時レイヤを作成する

```
from qgis.core import QgsFeatureRequest

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

ジオメトリを取得する

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

```
10.000000, 10.000000
```

ジオメトリを移動する

```
from qgis.core import QgsFeature, QgsGeometry
poly = QgsFeature()
geom = QgsGeometry.fromWkt("POINT(7 45)")
geom.translate(1, 1)
poly.setGeometry(geom)
print(poly.geometry())
```

```
<QgsGeometry: Point (8 46)>
```

CRS を設定する

```
from qgis.core import QgsProject, QgsCoordinateReferenceSystem

for layer in QgsProject.instance().mapLayers().values():
    layer.setCrs(QgsCoordinateReferenceSystem('EPSG:4326'))
```

CRS を確認する

```
from qgis.core import QgsProject

for layer in QgsProject.instance().mapLayers().values():
    crs = layer.crs().authid()
    layer.setName('{} ({})'.format(layer.name(), crs))
```

特定のフィールド列を隠す

```
from qgis.core import QgsEditorWidgetSetup

def fieldVisibility (layer,fname):
    setup = QgsEditorWidgetSetup('Hidden', {})
    for i, column in enumerate(layer.fields()):
        if column.name()==fname:
            layer.setEditorWidgetSetup(idx, setup)
            break
        else:
            continue
```

WKT からレイヤ

```
from qgis.core import QgsVectorLayer, QgsFeature, QgsGeometry, QgsProject

layer = QgsVectorLayer('Polygon?crs=epsg:4326', 'Mississippi', 'memory')
pr = layer.dataProvider()
poly = QgsFeature()
geom = QgsGeometry.fromWkt("POLYGON ((-88.82 34.99,-88.09 34.89,-88.39 30.34,-89.57 30.18,-89.73 31,-91.63 30.99,-90.87 32.37,-91.23 33.44,-90.93 34.23,-90.30 34.99,-88.82 34.99))")
poly.setGeometry(geom)
pr.addFeatures([poly])
layer.updateExtents()
QgsProject.instance().addMapLayers([layer])
```

GeoPackage から全てのレイヤを読み込む

```
from qgis.core import QgsDataProvider

fileName = "testdata/sublayers.gpkg"
layer = QgsVectorLayer(fileName, "test", "ogr")
subLayers = layer.dataProvider().subLayers()

for subLayer in subLayers:
    name = subLayer.split(QgsDataProvider.SUBLAYER_SEPARATOR)[1]
    uri = "%s|layername=%s" % (fileName, name,)
    # Create layer
    sub_vlayer = QgsVectorLayer(uri, name, 'ogr')
    # Add layer to map
    QgsProject.instance().addMapLayer(sub_vlayer)
```

タイルレイヤ (XYZ-Layer) を読み込む

```
from qgis.core import QgsRasterLayer, QgsProject

def loadXYZ(url, name):
    rasterLyr = QgsRasterLayer("type=xyz&url=" + url, name, "wms")
    QgsProject.instance().addMapLayer(rasterLyr)

urlWithParams = 'https://tile.openstreetmap.org/%7Bz%7D/%7Bx%7D/%7By%7D.png&zmax=19&zmin=0&crs=EPSG3857'
loadXYZ(urlWithParams, 'OpenStreetMap')
```

すべてのレイヤを削除する

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

すべてを削除する

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

21.7 目次

チェックの入ったレイヤにアクセスする

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

コンテキストメニューを削除する

```
ltv = iface.layerTreeView()
mp = ltv.menuProvider()
ltv.setMenuProvider(None)
# Restore
ltv.setMenuProvider(mp)
```

21.8 拡張 TOC

Root ノード

```
from qgis.core import QgsVectorLayer, QgsProject, QgsLayerTreeLayer

root = QgsProject.instance().layerTreeRoot()
node_group = root.addGroup("My Group")

layer = QgsVectorLayer("Point?crs=EPSG:4326", "layer name you like", "memory")
QgsProject.instance().addMapLayer(layer, False)

node_group.addLayer(layer)

print(root)
print(root.children())
```

最初の子ノードにアクセスする

```
from qgis.core import QgsLayerTreeGroup, QgsLayerTreeLayer, QgsLayerTree

child0 = root.children()[0]
print (child0.name())
print (type(child0))
print (isinstance(child0, QgsLayerTreeLayer))
print (isinstance(child0.parent(), QgsLayerTree))
```

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

グループとそのノードを取得する

```
from qgis.core import QgsLayerTreeGroup, QgsLayerTreeLayer

def get_group_layers(group):
   print('- group: ' + group.name())
   for child in group.children():
      if isinstance(child, QgsLayerTreeGroup):
         # Recursive call to get nested groups
         get_group_layers(child)
      else:
         print('  - layer: ' + child.name())


root = QgsProject.instance().layerTreeRoot()
for child in root.children():
   if isinstance(child, QgsLayerTreeGroup):
      get_group_layers(child)
   elif isinstance(child, QgsLayerTreeLayer):
      print ('- layer: ' + child.name())
```

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

名前からグループを取得する

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

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

ID からレイヤを取得する

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

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

レイヤを追加する

```
from qgis.core import QgsVectorLayer, QgsProject

layer1 = QgsVectorLayer("Point?crs=EPSG:4326", "layer name you like 2", "memory")
```

(次のページに続く)

(前のページからの続き)

```
QgsProject.instance().addMapLayer(layer1, False)
node_layer1 = root.addLayer(layer1)
# Remove it
QgsProject.instance().removeMapLayer(layer1)
```

グループを追加する

```
from qgis.core import QgsLayerTreeGroup

node_group2 = QgsLayerTreeGroup("Group 2")
root.addChildNode(node_group2)
QgsProject.instance().mapLayersByName("layer name you like")[0]
```

読み込んだレイヤを移動する

```
layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
root = QgsProject.instance().layerTreeRoot()

myLayer = root.findLayer(layer.id())
myClone = myLayer.clone()
parent = myLayer.parent()

myGroup = root.findGroup("My Group")
# Insert in first position
myGroup.insertChildNode(0, myClone)

parent.removeChildNode(myLayer)
```

読み込んだレイヤを特定のグループに移動する

```
QgsProject.instance().addMapLayer(layer, False)

root = QgsProject.instance().layerTreeRoot()
myGroup = root.findGroup("My Group")
myOriginalLayer = root.findLayer(layer.id())
myLayer = myOriginalLayer.clone()
myGroup.insertChildNode(0, myLayer)
parent.removeChildNode(myOriginalLayer)
```

アクティブレイヤの可視設定を切り替える

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

あるグループが選択されているかどうか

```
def isMyGroupSelected( groupName ):
    myGroup = QgsProject.instance().layerTreeRoot().findGroup( groupName )
    return myGroup in iface.layerTreeView().selectedNodes()

print(isMyGroupSelected( 'my group name' ))
```

```
False
```

ノードを展開する/折り畳む

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

隠れたノード技

```
from qgis.core import QgsProject

model = iface.layerTreeView().layerTreeModel()
ltv = iface.layerTreeView()
root = QgsProject.instance().layerTreeRoot()

layer = QgsProject.instance().mapLayersByName('layer name you like')[0]
node = root.findLayer(layer.id())

index = model.node2index( node )
ltv.setRowHidden( index.row(), index.parent(), True )
node.setCustomProperty( 'nodeHidden', 'true')
ltv.setCurrentIndex(model.node2index(root))
```

ノードシグナル

```
def onWillAddChildren(node, indexFrom, indexTo):
    print ("WILL ADD", node, indexFrom, indexTo)

def onAddedChildren(node, indexFrom, indexTo):
    print ("ADDED", node, indexFrom, indexTo)

root.willAddChildren.connect(onWillAddChildren)
root.addedChildren.connect(onAddedChildren)
```

レイヤを削除する

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

グループを削除する

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

新しい目次 (TOC) を作る

```
root = QgsProject.instance().layerTreeRoot()
model = QgsLayerTreeModel(root)
view = QgsLayerTreeView()
view.setModel(model)
view.show()
```

ノードを移動する

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

ノード名を変更する

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

21.9 プロセシングアルゴリズム

アルゴリズムの一覧を取得する

```
from qgis.core import QgsApplication

for alg in QgsApplication.processingRegistry().algorithms():
    if 'buffer' == alg.name():
        print("{}:{} --> {}".format(alg.provider().name(), alg.name(), alg.
↪displayName()))
```

```
QGIS (native c++):buffer --> Buffer
```

アルゴリズムのヘルプを取得する

ランダム選択

```
from qgis import processing
processing.algorithmHelp("native:buffer")
```

```
...
```

アルゴリズムを実行する

この例では、プロジェクトに追加された一時メモリレイヤに結果が格納される。

```
from qgis import processing
result = processing.run("native:buffer", {'INPUT': layer, 'OUTPUT': 'memory:'})
QgsProject.instance().addMapLayer(result['OUTPUT'])
```

```
Processing(0): Results: {'OUTPUT': 'output_d27a2008_970c_4687_b025_f057abbd7319'}
```

アルゴリズムはいくつあるか？

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

プロバイダはいくつあるか？

```
from qgis.core import QgsApplication

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

式はいくつあるか？

```
from qgis.core import QgsExpression

len(QgsExpression.Functions())
```

21.10 装飾類

著作権

```
from qgis.PyQt.Qt import QTextDocument
from qgis.PyQt.QtGui import QFont

mQFont = "Sans Serif"
mQFontsize = 9
mLabelQString = "© QGIS 2019"
mMarginHorizontal = 0
mMarginVertical = 0
mLabelQColor = "#FF0000"

INCHES_TO_MM = 0.0393700787402 # 1 millimeter = 0.0393700787402 inches
case = 2

def add_copyright(p, text, xOffset, yOffset):
    p.translate( xOffset , yOffset  )
    text.drawContents(p)
    p.setWorldTransform( p.worldTransform() )

```

(次のページに続く)

(前のページからの続き)

```
def _on_render_complete(p):
    deviceHeight = p.device().height() # Get paint device height on which this painter is currently painting
    deviceWidth  = p.device().width() # Get paint device width on which this painter is currently painting
    # Create new container for structured rich text
    text = QTextDocument()
    font = QFont()
    font.setFamily(mQFont)
    font.setPointSize(int(mQFontsize))
    text.setDefaultFont(font)
    style = "<style type=\"text/css\"> p {color: " + mLabelQColor + "}</style>"
    text.setHtml( style + "<p>" + mLabelQString + "</p>" )
    # Text Size
    size = text.size()

    # RenderMillimeters
    pixelsInchX  = p.device().logicalDpiX()
    pixelsInchY  = p.device().logicalDpiY()
    xOffset  = pixelsInchX  * INCHES_TO_MM * int(mMarginHorizontal)
    yOffset  = pixelsInchY  * INCHES_TO_MM * int(mMarginVertical)

    # Calculate positions
    if case == 0:
        # Top Left
        add_copyright(p, text, xOffset, yOffset)

    elif case == 1:
        # Bottom Left
        yOffset = deviceHeight - yOffset - size.height()
        add_copyright(p, text, xOffset, yOffset)

    elif case == 2:
        # Top Right
        xOffset  = deviceWidth  - xOffset - size.width()
        add_copyright(p, text, xOffset, yOffset)

    elif case == 3:
        # Bottom Right
        yOffset  = deviceHeight - yOffset - size.height()
        xOffset  = deviceWidth  - xOffset - size.width()
        add_copyright(p, text, xOffset, yOffset)

    elif case == 4:
        # Top Center
```

(次のページに続く)

(前のページからの続き)

```
        xOffset = deviceWidth / 2
        add_copyright(p, text, xOffset, yOffset)

    else:
        # Bottom Center
        yOffset = deviceHeight - yOffset - size.height()
        xOffset = deviceWidth / 2
        add_copyright(p, text, xOffset, yOffset)

# Emitted when the canvas has rendered
iface.mapCanvas().renderComplete.connect(_on_render_complete)
# Repaint the canvas map
iface.mapCanvas().refresh()
```

21.11 コンポーザー

印刷レイアウトを名前で取得する

```
composerTitle = 'MyComposer' # Name of the composer

project = QgsProject.instance()
projectLayoutManager = project.layoutManager()
layout = projectLayoutManager.layoutByName(composerTitle)
```

21.12 出典

- QGIS Python (PyQGIS) API
- QGIS C++ API
- StackOverFlow QGIS questions
- Script by Klas Karlsson