PyQGIS 3.40 developer cookbook

QGIS Project

2026 년 02 월 16 일

Contents

1 소개 **3**
1.1 파이썬 콘솔에서의 스크립트 작업 . . . 4
1.2 파이썬 플러그인 . . . 4
1.3 QGIS 구동 시 파이썬 코드 실행 . . . 5
1.4 파이썬 응용 프로그램 . . . 6
1.5 PyQt 및 SIP 에 대한 기술 노트 . . . 9

2 프로젝트 불러오기 **11**
2.1 경로 오류 해결하기 . . . 12
2.2 속도 향상을 위해 플래그 사용하기 . . . 13

3 레이어 불러오기 **15**
3.1 벡터 레이어 . . . 15
3.2 래스터 레이어 . . . 18
3.3 QgsProject 인스턴스 . . . 21

4 TOC(Table Of Contents) 에 접근하기 **23**
4.1 QgsProject 클래스 . . . 23
4.2 QgsLayerTreeGroup 클래스 . . . 24

5 래스터 레이어 사용하기 **27**
5.1 레이어 상세 정보 . . . 27
5.2 렌더링 작업자 . . . 28
5.3 값 쿼리하기 . . . 30
5.4 래스터 데이터 편집하기 . . . 30

6 벡터 레이어 사용하기 **33**
6.1 속성에 대한 정보 검색하기 . . . 34
6.2 벡터 레이어 작업 반복하기 . . . 35
6.3 피처 선택하기 . . . 36
6.4 벡터 레이어 수정하기 . . . 38
6.5 공간 인덱스 사용하기 . . . 42
6.6 QgsVectorLayerUtils 클래스 . . . 43
6.7 벡터 레이어 생성하기 . . . 43
6.8 벡터 레이어의 표현 (심볼) . . . 48
6.9 남은 이야기들 . . . 57

7 도형 다루기 **59**
7.1 도형 작성하기 . . . 60
7.2 도형에 접근하기 . . . 60
7.3 도형 술어 및 연산 . . . 62

8 투영체 지원 **65**
8.1 좌표계 . . . 65
8.2 좌표계 변환 . . . 67

9 맵 캔버스 사용하기 **69**
9.1 맵 캔버스 내장시키기 . . . 70
9.2 고무줄과 꼭짓점 마커 . . . 71
9.3 캔버스에서 맵 도구 사용하기 . . . 72
9.4 사용자 정의 맵 도구 작성하기 . . . 74
9.5 사용자 정의 맵 캔버스 항목 작성하기 . . . 75

10 맵 렌더링 및 출력하기 **77**
10.1 단순 렌더링 . . . 78
10.2 서로 다른 좌표계를 가진 레이어들을 렌더링하기 . . . 79
10.3 인쇄 조판을 사용해서 산출하기 . . . 79

11 값을 필터링하고 계산하는 표현식 **83**
11.1 표현식 파싱하기 . . . 84
11.2 표현식 평가하기 . . . 85
11.3 표현식 오류 처리하기 . . . 87

12 읽기 및 저장하기 설정 **89**

13 사용자에게 정보 전달하기 **93**
13.1 메시지 보이기: QgsMessageBar 클래스 . . . 94
13.2 진행률 보이기 . . . 97
13.3 로그 작성하기 . . . 97

14 인증 인프라스트럭처 **101**
14.1 소개 . . . 102
14.2 용어 해설 . . . 102
14.3 QgsAuthManager: 입구점 . . . 103
14.4 플러그인이 인증 인프라스트럭처를 사용하도록 조정하기 . . . 106
14.5 인증 GUI . . . 107

15 태스크 - 배경에서 무거운 작업 하기 **111**
15.1 소개 . . . 111
15.2 예제 . . . 113

16 파이썬 플러그인 개발하기 **119**
16.1 파이썬 플러그인 구성하기 . . . 119
16.2 코드 조각 . . . 129
16.3 플러그인 작성 및 디버그 작업을 위한 IDE 설정 . . . 133
16.4 사용자 플러그인 배포하기 . . . 141

17 공간 처리 플러그인 작성하기 **145**
17.1 처음부터 생성하기 . . . 145
17.2 플러그인을 업데이트하기 . . . 146
17.3 Implementing custom Processing algorithms . . . 147

18 플러그인 레이어 사용하기 **151**
18.1 QgsPluginLayer 하위 클래스 만들기 . . . 151

19 네트워크 분석 라이브러리 **155**
19.1 일반 정보 . . . 155
19.2 그래프 작성하기 . . . 156
19.3 그래프 분석 . . . 158

20 QGIS 서버와 파이썬 **165**
20.1 소개 . . . 165
20.2 서버 API 기본 사항 . . . 166
20.3 독립형 또는 내장형 . . . 166
20.4 서버 플러그인 . . . 167

21 PyQGIS 의 꼼수와 편법 **179**
21.1 사용자 인터페이스 . . . 179
21.2 설정 . . . 180
21.3 툴바 . . . 180
21.4 메뉴 그룹 . . . 180
21.5 캔버스 . . . 181
21.6 레이어 . . . 181
21.7 TOC(Table of Contents) . . . 185
21.8 고급 TOC . . . 185
21.9 공간 처리 알고리즘 . . . 188
21.10 장식자 (decorator) . . . 189
21.11 작성자 (composer) . . . 191
21.12 소스 . . . 191

CHAPTER 1

소개

이 문서는 교재와 참조 지침서 둘 다를 목표로 작성되었습니다. 이 문서가 가능한 모든 사용례를 모두 보여주지는 못하지만, 중요한 기능들을 살펴보기엔 충분할 것입니다.

자유 소프트웨어 재단이 발행한 버전 1.3 이상의 GNU 자유 문서 사용 허가서의 조건에 따라 이 문서를 장절항목 변경 없이, 앞표지 및 뒤표지 텍스트 추가 없이 복사, 배포, 그리고/또는 수정할 수 있습니다.

이 사용 허가서의 복사본은 부록 gnu_fdl 에 포함되어 있습니다.

이 사용 허가서는 이 문서에 있는 모든 코드 조각들에도 적용됩니다.

파이썬을 처음 지원하기 시작한 것은 QGIS 0.9 버전부터입니다. QGIS 데스크탑에서 파이썬을 사용할 수 있는 몇 가지 방법 (을 다음 절들에서 설명합니다) 이 있습니다:

- QGIS 안에 있는 파이썬 콘솔에서 명령어 실행
- 플러그인 생성 및 사용
- QGIS 구동 시 파이썬 코드 자동 실행
- 공간 처리 알고리즘 생성
- QGIS 에서 표현식 용 함수 생성
- QGIS API 를 바탕으로 사용자 정의 응용 프로그램 생성

QGIS 서버에 대해서도 파이썬 바인딩을 사용할 수 있습니다. 이 사용법은 파이썬 플러그인 (*QGIS 서버와 파이썬* 참조) 과 파이썬 응용 프로그램에 QGIS 서버를 내장시키기 위해 사용할 수 있는 파이썬 바인딩을 포함합니다.

QGIS 라이브러리 클래스를 문서화한 완전한 QGIS C++ API 참조가 있습니다. 파이썬의 QGIS API (pyqgis) 는 C++ API 와 거의 동일합니다.

흔한 작업들을 수행하는 방법을 배우는 데 좋은 또다른 리소스는 플러그인 저장소 에서 기존 플러그인을 다운로드해서 그 코드를 공부하는 것입니다.

1.1 파이썬 콘솔에서의 스크립트 작업

QGIS 는 스크립트 작업을 위한 통합 파이썬 콘솔 을 제공합니다. *Plugins* ⸮ *Python Console* 메뉴 항목을 클릭해서 열 수 있습니다.

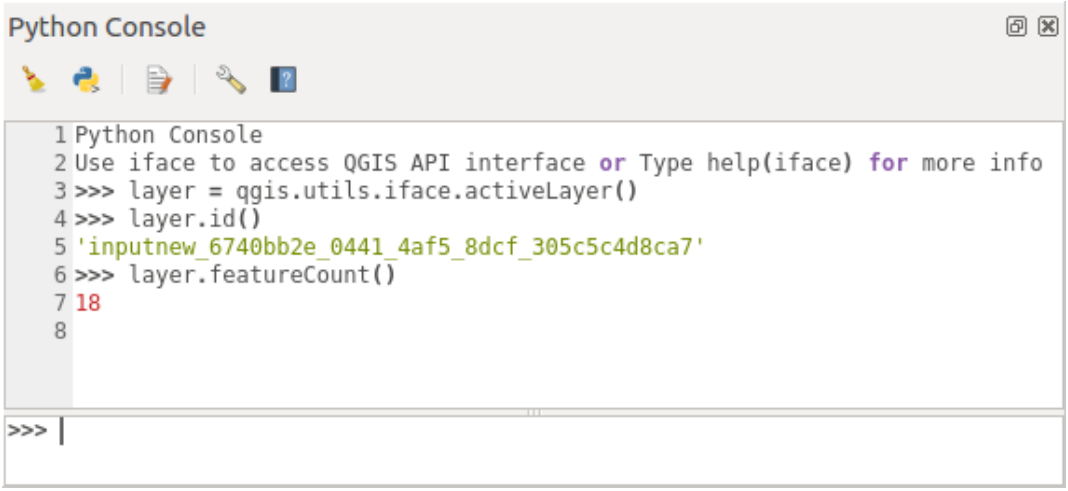


그림 1.1: QGIS 파이썬 콘솔

앞의 스크린샷은 현재 레이어 목록에서 선택한 레이어를 가져오는 방법과, 레이어 ID 를 보는 방법, 그리고 벡터 레이어인 경우 피처 개수를 보는 선택적인 방법을 보여주고 있습니다. QGIS 환경과 쌍방향으로 작업하기 위한, `QgisInterface` 클래스의 인스턴스인 `iface` 변수가 있습니다. 이 인터페이스를 통해 QGIS 응용 프로그램의 맵 캔버스, 메뉴, 툴바, 그리고 기타 부분들에 접근할 수 있습니다.

사용자 편의성을 위해, 콘솔이 시작될 때 다음 선언문이 실행됩니다. (향후 더 많은 초기 명령어를 설정할 수 있게 될 것입니다.)

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

이 콘솔을 자주 사용하는 사용자라면 (*Settings* ⸮ *Keyboard shortcuts*…메뉴에서) 콘솔을 여는 단축키를 설정해두는 편이 유용할 수도 있습니다.

1.2 파이썬 플러그인

플러그인을 사용하면 QGIS 의 기능을 확장할 수 있습니다. 플러그인은 파이썬으로 작성할 수 있습니다. 파이썬 플러그인이 C++ 플러그인을 뛰어넘는 주요한 장점은 배포가 단순하다는 점과 (각 플랫폼 별로 컴파일해야 할 필요가 없습니다) 개발이 더 쉽다는 점입니다.

파이썬을 지원하기 시작한 후로 여러 기능을 커버하는 수많은 플러그인이 작성돼 왔습니다. 플러그인 설치자는 사용자가 파이썬 플러그인을 쉽게 가져오고, 업그레이드하고, 제거할 수 있게 해줍니다. 플러그인과 플러그인 개발에 대해 더 많이 알고 싶다면 파이썬 플러그인 페이지를 참조하세요.

파이썬으로 플러그인을 생성하는 일은 매우 간단합니다. 자세한 내용은파이썬 플러그인 개발하기 강의를 참조하십시오.

참고

QGIS 서버 용 파이썬 플러그인도 사용할 수 있습니다. 더 자세한 내용은*QGIS 서버와 파이썬* 을 참조하세요.

1.2.1 공간 처리 플러그인

공간 처리 플러그인은 데이터를 공간 처리할 수 있습니다. 공간 처리 플러그인은 파이썬 플러그인보다 더 개발하기 쉽고, 더 특정 목적에 가깝고, 더 가볍습니다. 공간 처리 플러그인 작성하기 에서 어떤 경우에 공간 처리 알고리즘을 사용해야 알맞은지 그리고 어떻게 개발해야 하는지를 설명하고 있습니다.

1.3 QGIS 구동 시 파이썬 코드 실행

QGIS 를 구동할 때마다 파이썬 코드를 실행시킬 수 있는 서로 다른 방법들이 있습니다.

1. startup.py 스크립트 생성하기
2. 기존 파이썬 파일에 `PYQGIS_STARTUP` 환경 변수 설정하기
3. `--code init_qgis.py` 파라미터를 사용해서 구동 스크립트 지정하기

1.3.1 `startup.py` 파일

QGIS 를 구동할 때마다, 사용자의 파이썬 홈 디렉터리와 시스템 경로 목록에서 `startup.py` 라는 파일을 검색합니다. 해당 파일이 존재하는 경우, 내장 파이썬 해석기가 이 파일을 실행합니다.

사용자 홈 디렉터리 경로는 보통 다음 아래에서 찾을 수 있습니다:

- 리눅스: `.local/share/QGIS/QGIS3`
- 윈도우: `AppData\Roaming\QGIS\QGIS3`
- 맥 OS: `Library/Application Support/QGIS/QGIS3`

기본 시스템 경로는 운영체제에 따라 달라집니다. 사용자에 맞게 작동하는 경로를 찾으려면 파이썬 콘솔을 열고 `QStandardPaths.standardLocations(QStandardPaths.AppDataLocation)` 를 실행하십시오. 기본 디렉터리 목록이 출력될 것입니다.

QGIS 응용 프로그램의 구동 초기에, QGIS 에서 파이썬이 초기 설정되는 즉시 `startup.py` 스크립트를 실행합니다.

1.3.2 PYQGIS_STARTUP 환경 변수

`PYQGIS_STARTUP` 환경 변수를 기존 파이썬 파일의 경로로 설정하면 QGIS 초기 설정이 완료되기 직전에 파이썬 코드를 실행시킬 수 있습니다.

이 코드는 QGIS 초기 설정이 완료되기 전에 실행될 것입니다. 이 방법은 원하지 않는 경로를 가지고 있을 지도 모를 sys.path 를 지우는 데, 또는 가상 환경을 설정할 필요 없이, 예를 들어 맥에서 홈브루 또는 MacPorts 를 설치할 필요 없이 초기 환경을 격리하기/불러오기하는 데 매우 유용합니다.

1.3.3 --code 파라미터

사용자가 QGIS 에 구동 파라미터로 실행되는 사용자 정의 코드를 제공할 수 있습니다. 파이썬 파일을, 예를 들어 `qgis_init.py` 파일을 생성한 다음, 명령 줄에서 `qgis --code qgis_init.py` 명령어를 실행해서 QGIS 를 구동하면 됩니다.

`--code` 파라미터를 통해 제공된 코드는 QGIS 초기 설정 단계 후반에, 응용 프로그램 구성 요소들을 불러온 후에 실행됩니다.

1.3.4 파이썬 용 추가 인자들

사용자의 `--code` 스크립트 또는 실행되는 다른 파이썬 코드에 추가 인자를 더하고 싶은 경우, `--py-args` 인자를 사용하면 됩니다. `--py-args` 뒤에 그리고 (존재하는 경우) `--` 인자 앞에 오는 모든 인자는 파이썬으로 전달될 것이지만 QGIS 응용 프로그램 자체는 해당 인자를 무시할 것입니다.

다음 예시에서, `myfile.tif` 는 파이썬에서 `sys.argv` 를 통해 사용할 수 있지만 QGIS 가 `myfile.tif` 를 불러오지는 않을 것입니다. 반면 QGIS 는 `otherfile.tif` 를 불러올 것이지만 `sys.argv` 에는 없습니다.

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

파이썬 안에서 나온 모든 명령 줄 파라미터에 접근하고 싶은 경우, `QCoreApplication.arguments()` 를 사용하면 됩니다.

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

1.4 파이썬 응용 프로그램

공간 처리 자동화를 위한 스크립트를 작성하는 것이 편리한 경우가 많습니다. PyQGIS 를 사용하면 이를 완벽하게 작업할 수 있습니다—qgis.core 모듈을 가져와서 초기 설정을 하기만 하면 공간 처리 작업을 할 준비가 끝납니다.

또는 사용자가 GIS 기능을 사용하는 대화형 응용 프로그램을 만들려 할 수도 있습니다—측정 작업을 수행하거나, 맵을 PDF 로 내보내거나, 등등과 같은 기능 말이죠. qgis.gui 모듈은 여러 GUI 구성 요소를 제공합니다. 가장 주목할 만한 요소는 맵 캔버스 위젯으로, 응용 프로그램에 통합되어 확대/축소, 이동, 그리고/또는 다른 사용자 정의 맵 도구들을 지원합니다.

PyQGIS 사용자 정의 응용 프로그램 또는 독립 실행형 스크립트는 투영체 정보와 벡터 및 래스터 레이어를 읽어오기 위한 제공자 같은 QGIS 리소스의 정확한 위치를 찾을 수 있도록 반드시 환경설정해줘야만 합니다. 사용자 응용 프로그램 또는 스크립트의 시작 부분에 몇 줄을 추가해주면 QGIS 리소스를 초기 설정할 수 있습니다. 사용자 정의 응용 프로그램 또는 독립 실행형 스크립트를 위해 QGIS 를 초기 설정해주는 코드는 비슷합니다. 다음은 그 각각의 예시입니다.

> **참고**
>
> 사용자 스크립트에 `qgis.py` 라는 파일 이름을 사용하지 마십시오. 스크립트의 파일 이름이 바인딩을 가릴 것이기 때문에 파이썬이 바인딩을 가져올 수 없을 것입니다.

1.4.1 독립 실행형 스크립트에서 PyQGIS 사용하기

독립 실행형 스크립트를 시작하려면, 스크립트 시작 부분에 QGIS 리소스를 초기 설정하십시오:

```
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()
```

먼저 qgis.core 모듈을 가져온 다음 앞에 붙는 (prefix) 경로를 환경설정합니다. 이 접두 경로는 QGIS 가 사용자 시스템 상에 설치돼 있는 위치입니다. 스크립트에서 `setPrefixPath()` 메소드를 호출해서 접두 경로를 환경설정합니다. `setPrefixPath()` 의 두 번째 인자를 `True` 로 설정해서 기본 경로들을 사용한다고 지정합니다.

QGIS 의 설치 경로는 플랫폼에 따라 다릅니다. 사용자 시스템에서 설치 경로를 찾는 가장 쉬운 방법은 QGIS 안에서파이썬 콘솔에서의 스크립트 작업 을 사용해서 다음을 실행한 다음 출력물을 살펴보는 것입니다:

```
QgsApplication.prefixPath()
```

접두 경로를 환경설정한 다음, `qgs` 변수에 `QgsApplication` 클래스를 가리키는 참조를 저장합니다. 두 번째 인자를 `False` 로 설정해서 GUI 를 사용할 계획이 없다고 지정합니다. 우리는 지금 독립 실행형 스크립트를 작성하고 있기 때문이죠. `QgsApplication` 클래스를 환경설정한 다음, `initQgis()` 메소드를 호출해서 QGIS 데이터 제공자와 레이어 레지스트리를 불러옵니다.

```
qgs.initQgis()
```

QGIS 를 초기 설정하고 나면, 스크립트의 다음 부분을 작성할 준비가 끝납니다. 마지막으로 `exitQgis()` 메소드를 호출해서 메모리로부터 데이터 제공자와 레리어 레지스트리를 제거하는 것으로 마무리합니다.

```
qgs.exitQgis()
```

1.4.2 사용자 정의 응용 프로그램에서 PyQGIS 사용하기

독립 실행형 스크립트에서 *PyQGIS* 사용하기 와 사용자 정의 PyQGIS 응용 프로그램의 유일한 차이점은 `QgsApplication` 클래스를 인스턴스화할 때의 두 번째 인자뿐입니다. `False` 대신 `True` 를 전달해서 GUI 를 사용할 계획이라고 나타내십시오.

```
from qgis.core import *

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

# 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()
```

이제 QGIS API 를 작업할 수 있습니다—레이어를 불러와서 어떤 공간 처리를 하거나 GUI 를 맵 캔버스와 함께 열거나, 가능성은 무궁무진합니다. :-)

1.4.3 사용자 정의 응용 프로그램 실행하기

QGIS 라이브러리 및 알맞은 파이썬 모듈이 잘 알려진 위치에 없는 경우, 사용자 시스템에 이들을 검색해야 할 위치를 알려줘야 합니다—그렇지 않으면 파이썬이 불평할 것입니다:

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

`PYTHONPATH` 환경 변수를 설정하면 이 문제를 해결할 수 있습니다. 다음 명령어들에서, `<qgispath>` 를 사용자의 실제 QGIS 설치 경로로 대체해야 할 것입니다:

- 리눅스: **`export PYTHONPATH=/<qgispath>/share/qgis/python`**
- 윈도우: **`set PYTHONPATH=c:\<qgispath>\python`**
- 맥 OS: **`export PYTHONPATH=/<qgispath>/Contents/Resources/python`**

이제 PyQGIS 모듈들을 가리키는 경로를 지정했지만, 이 모듈들은 `qgis_core` 와 `qgis_gui` 라이브러리에 의존하고 있습니다. (파이썬 모듈은 래퍼 (wrapper) 역할만 할 뿐입니다.) 운영체제가 이 라이브러리들을 가리키는 경로를 모르고 있을 수도 있기 때문에, 이 경우 다시 가져오기 오류 메시지를 보게 될 것입니다. (이 메시지는 사용자 시스템에 따라 달라질 수도 있습니다.):

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

동적 링커 (dynamic linker) 의 검색 경로에 QGIS 라이브러리가 있는 디렉터리를 추가해서 이 문제를 해결하십시오:

- 리눅스: **`export LD_LIBRARY_PATH=/<qgispath>/lib`**
- 윈도우: **`set PATH=C:\<qgispath>\bin;C:\<qgispath>\apps\<qgisrelease>\bin;%PATH%`**, 이때 `<qgisrelease>` 를 사용자가 대상으로 하는 배포판 유형 (예: `qgis-ltr`, `qgis`, `qgis-dev`) 으로 대체해야 할 것입니다.

이러한 명령어를 구동 과정을 처리할 부트스트랩 (bootstrap) 스크립트에 넣어 둘 수 있습니다. PyQGIS 를 이용하는 사용자 정의 응용 프로그램을 배포하는 경우, 보통 다음 두 가지 방법을 쓸 수 있습니다:

- 사용자 응용 프로그램을 설치하기 전에 필수적으로 QGIS 를 설치해야 한다고 요청하는 방법입니다. 응용 프로그램 설치자가 QGIS 라이브러리의 기본 위치를 검색하게 하고, 찾지 못했을 경우 사용자가 경로를 설정할 수 있도록 해줘야 합니다. 이 방법은 좀 더 간단하다는 장점이 있지만, 사용자가 더 많은 단계를 거쳐야 합니다.
- QGIS 를 응용 프로그램과 함께 패키징하는 방법입니다. 응용 프로그램을 배포하는 데 더 많은 노력이 필요하고 용량이 더 커지게 되지만, 사용자가 추가로 다른 소프트웨어를 다운로드해 설치해야 하는 부담이 줄어들게 됩니다.

이 두 가지 배포 모델을 섞을 수도 있습니다. 윈도우와 맥 OS 에서는 독립 설치형 응용 프로그램을 제공하지만, 리눅스에서는 사용자와 사용자의 패키지 관리자에 GIS 설치를 맡기도록 할 수 있습니다.

1.5 PyQt 및 SIP 에 대한 기술 노트

우리는 스크립트 작업에 가장 선호되는 언어 가운데 하나이기 때문에 파이썬을 사용하기로 결정했습니다. QGIS 3 버전의 PyQGIS 바인딩은 SIP 및 PyQt5 에 의존하고 있습니다. 좀 더 널리 쓰이는 SWIG 대신 SIP 를 사용하는 이유는 QGIS 코드가 Qt 라이브러리에 의존하고 있기 때문입니다. SIP 를 사용해서 Qt(PyQt) 에 파이썬을 바인딩하므로 PyQGIS 를 PyQt 와 완벽하게 통합할 수 있는 것입니다.

CHAPTER 2

프로젝트 불러오기

힌트

PyQGIS 콘솔을 사용하지 않는 경우 이 페이지에 있는 코드 조각들을 다음과 같이 가져와야 합니다:

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

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

플러그인으로부터 기존 프로젝트를, 또는 (더 자주) 독립 설치형 QGIS 파이썬 응용 프로그램을 개발할 때, 기존 프로젝트를 불러와야 하는 경우가 많습니다. (파이썬 응용 프로그램 참조)

현재 QGIS 응용 프로그램으로 프로젝트를 불러오려면 `QgsProject` 클래스의 인스턴스를 생성해야 합니다. 이 클래스는 싱글턴 (singleton) 클래스이기 때문에, 생성하려면 이 클래스의 `instance()` 메소드를 사용해야만 합니다. 이 클래스의 `read()` 메소드를 호출해서 불러올 프로젝트의 경로를 넘겨줄 수 있습니다:

```
# 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
```

프로젝트를 (예를 들어 몇몇 레이어를 추가하거나 제거하기 위해) 수정하고 변경 사항을 저장해야 한다면, 프로젝트 인스턴스의 `write()` 메소드를 호출하십시오. `write()` 메소드도 프로젝트를 새 위치에 저장하기 위한 선택적인 경로를 입력받습니다.

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

`read()` 와 `write()` 두 함수 모두 작업 수행이 성공적이었는지 확인할 수 있는 불 (boolean) 값을 반환합니다.

> **참고**
>
> QGIS 독립 설치형 응용 프로그램을 작성하고 있는 경우, 불러온 프로젝트를 캔버스와 동기화시키려면 다음 예시에서처럼 `QgsLayerTreeMapCanvasBridge` 클래스의 인스턴스를 생성해야 합니다:
>
> ```
> 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 경로 오류 해결하기

프로젝트에 불러온 레이어들이 또다른 위치로 옮겨지는 경우가 있을 수 있습니다. 이런 프로젝트를 다시 불러오는 경우 모든 레이어 경로가 깨지게 됩니다. `QgsPathResolver` 클래스를 사용하면 프로젝트 안에서 레이어 경로를 다시 작성할 수 있습니다.

이 클래스의 `setPathPreprocessor()` 메소드는 사용자 정의 경로 전처리기 (pre-processor) 함수를 설정해서, 경로와 데이터 소스를 파일 참조 또는 레이어 소스로 해석하기 전에 조작할 수 있게 해줍니다.

처리기 (processor) 함수는 반드시 (원본 파일 경로 또는 데이터 소스를 표현하는) 단일 문자열 인자를 받아서 해당 경로를 처리한 버전을 반환해야만 합니다. 오류가 발생한 어떤 레이어 처리자 (handler) 보다도 **먼저** 이 경로 전처리기 함수를 호출합니다. 전처리기를 여러 개 설정한 경우, 원래 전처리기들이 설정된 순서대로 차례차례 호출할 것입니다.

다음은 몇몇 활용 사례입니다:

1. 변경된 경로를 대체:

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

   QgsPathResolver.setPathPreprocessor(my_processor)
   ```

2. 새 데이터베이스 호스트 주소로 대체:

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

   QgsPathResolver.setPathPreprocessor(my_processor)
   ```

3. 저장된 데이터베이스 인증 정보를 새 정보로 대체:

```
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)
```

마찬가지로, 경로 작성자 함수에 `setPathWriter()` 메소드를 사용할 수 있습니다.

경로를 변수로 대체하는 예시입니다:

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

QgsPathResolver.setPathWriter(my_processor)
```

두 메소드 모두 전처리기 또는 메소드가 추가한 작성자를 제거하는 데 사용할 수 있는 `id` 를 반환합니다. `removePathPreprocessor()` 및 `removePathWriter()` 를 참조하세요.

2.2 속도 향상을 위해 플래그 사용하기

완전하게 기능하는 프로젝트를 사용할 필요는 없고 특정한 이유 때문에 프로젝트에 접근만 하면 되는 경우, 플래그가 도움이 될 수도 있습니다. `ProjectReadFlag` 클래스 아래에서 전체 플래그 목록을 찾아볼 수 있습니다. 플래그 여러 개를 함께 추가할 수도 있습니다.

예를 들어 실제 레이어와 데이터에는 관심이 없고 (예를 들면 조판 또는 3D 뷰 설정 때문에) 단순히 프로젝트에 접근만 하고 싶은 경우, 데이터 무결성 검증 단계를 뛰어넘고 오류가 발생한 레이어 대화창이 열리는 일을 방지하려면 `DontResolveLayers` 플래그를 다음 코드처럼 사용하면 됩니다:

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

플래그를 더 많이 추가하려면 반드시 파이썬 비트 연산자 (bitwise operator) OR(`|`) 를 사용해야만 합니다.

CHAPTER 3

레이어 불러오기

힌트

이 페이지에 있는 코드 조각들을 다음과 같이 가져와야 합니다:

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

데이터를 담고 있는 레이어를 열어봅시다. QGIS 는 벡터 및 래스터 레이어를 인식합니다. 추가적으로 사용자 정의 레이어 유형도 사용할 수 있지만, 이 문서에서는 다루지 않을 것입니다.

3.1 벡터 레이어

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 벡터 레이어의 경우:

```
# 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!")
```

이렇게 하면 한번에 새 레이어를 생성해서 현재 QGIS 프로젝트에 추가합니다. (레이어 목록에 새 레이어를 나타나게 합니다.) 이 함수는 레이어 인스턴스를 반환하거나 레이어를 불러올 수 없는 경우 `None` 을 반환합니다.

다음 목록은 벡터 데이터 제공자들을 통해 여러 데이터 소스에 접근하는 방법을 보여줍니다:

- 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)
    ```

    Note that if your GeoPackage contains relationships between layers. Simply using `addMapLayer` does not automatically create these relationships between the layers in your QGIS project. Add the following code to automatically find the relationships in the database and add appropriate ones to the project properties.

    ```
    from qgis.core import QgsRelationManager

    project = QgsProject.instance()
    relation_manager = project.relationManager()
    existing_relations = list(relation_manager.relations().values())

    layers = [
        layer for layer in project.mapLayers().values() if layer.type() == layer.
    ↪VectorLayer
    ]

    discovered_relations = QgsRelationManager.discoverRelations(existing_
    ↪relations, layers)
    for relation in discovered_relations:
        relation_manager.addRelation(relation)
    ```

  - DXF 파일의 경우 (데이터 소스 URI 에 있는 내부 옵션들을 주목하십시오):

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

- PostGIS 데이터베이스—데이터 소스는 PostgreSQL 데이터베이스와의 연결을 생성하는 데 필요한 모든 정보를 가진 문자열입니다.

  `QgsDataSourceUri` 클래스가 이 문자열을 생성할 수 있습니다. QGIS 가 PostgreSQL 를 지원하도록 QGIS 를 컴파일해야 한다는 점을 기억하십시오. 그렇지 않으면 이 제공자를 사용할 수 없습니다:

```
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")
```

> **참고**
>
> `uri.uri(False)` 에 전달된 `False` 인자가 인증 환경설정 파라미터들이 확장되지 못하게 막습니다. 사용자가 어떤 인증 환경설정도 사용하지 않는 경우, 이 인자는 어떤 영향도 미치지 않습니다.

- CSV 파일 또는 기타 구분된 텍스트 파일들—쌍반점 (`;`) 을 구분자로 사용하고"x"필드에 X 좌표를 그리고"y" 필드에 Y 좌표를 가진 파일을 열려면 다음과 같은 코드를 사용해야 할 것입니다:

```
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)
```

> **참고**
>
> 제공자 문자열은 URL 로써 구성되어 있기 때문에, 경로 앞에 반드시 `file://` 을 붙여야만 합니다. 또 `x` 및 `y` 필드의 대체물로써 WKT(Well Known Text) 서식 도형을 사용할 수 있으며, 좌표계도 지정할 수 있습니다. 다음은 그 예시입니다:
>
> ```
> uri = "file:///some/path/file.csv?delimiter={}&crs=epsg:4723&wktField={}".
> ↪format(";", "shape")
> ```

- GPX 파일—"gpx"데이터 제공자는 GPX 파일에서 트랙, 루트, 웨이포인트를 읽어옵니다. GPX 파일을 열려면, 데이터 유형 (트랙/루트/웨이포인트) 을 URL 의 일부분으로 지정해줘야 합니다:

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

- SpatiaLite 데이터베이스—PostGIS 데이터베이스와 유사하게, `QgsDataSourceUri` 클래스를 사용해서 데이터 소스 식별자를 생성할 수 있습니다:

```
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)
```

- GDAL 을 통한 MySQL WKB 기반 도형—데이터 소스는 테이블과의 연결 문자열입니다:

```
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 연결—URI 와 `WFS` 제공자를 사용해서 연결을 정의합니다:

```
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")
```

표준 `urllib` 라이브러리를 사용해서 URI 를 생성할 수 있습니다:

```
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))
```

참고

다음 예시처럼 `QgsVectorLayer` 클래스 인스턴스에 대해 `setDataSource()` 메소드를 호출하면 기존 레이어의 데이터 소스를 변경할 수 있습니다:

```
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 래스터 레이어

래스터 파일에 접근하기 위해 GDAL 라이브러리를 사용합니다. GDAL 라이브러리는 광범위한 파일 포맷들을 지원합니다. 어떤 파일을 여는 데 문제가 있는 경우, 사용자의 GDAL 라이브러리가 그 특정 포맷을 지원하고 있는지 확인하십시오. (기본적으로 모든 포맷을 지원하지는 않습니다.) 파일로부터 래스터를 불러오려면, 래스터의 파일 이름과 표시 이름을 지정하십시오:

```
# 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 에서 래스터를 불러오려면:

```
# 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!")
```

벡터 레이어와 비슷하게, `QgisInterface` 클래스 객체의 addRasterLayer 함수를 사용해서 래스터 레이어를 불러올 수 있습니다:

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

이렇게 하면 한번에 새 레이어를 생성해서 현재 프로젝트에 추가합니다. (레이어 목록에 새 레이어를 나타나게 합니다.)

PostGIS 래스터를 불러오려면:

PostGIS 벡터와 비슷하게, URI 문자열을 사용해서 프로젝트에 PostGIS 래스터를 추가할 수 있습니다. 데이터베이스 연결 파라미터에 대해 재사용 가능한 문자열 목록 (dictionary) 을 유지하는 편이 효율적입니다. 이렇게 하면 적용 가능한 연결에 대한 목록을 쉽게 편집할 수 있습니다. 그 다음'postgresraster'제공자 메타데이터 객체를 사용해서 목록을 URI 로 인코딩합니다. 그러면 프로젝트에 래스터를 추가할 수 있습니다.

```
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.
    '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,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    '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")
```

WCS 서비스로부터도 래스터 레이어를 생성할 수 있습니다:

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

다음에서는 WCS URI 가 담을 수 있는 파라미터들을 설명합니다:

WCS URI 는 `&` 문자로 구분된 **키 = 값** 쌍으로 이루어집니다. 이는 URL 에 있는 동일한 방식으로 인코딩된 쿼리 문자열과 동일한 서식입니다. 특수 문자들이 제대로 인코딩되도록 보장하려면 `QgsDataSourceUri` 클래스를 사용해서 URI 를 작성해야 할 것입니다.

- **url** (필수): WCS 서버 URL 입니다. WCS 각 버전이 **GetCapabilities** 버전에 대해 서로 다른 파라미터 이름을 사용하고 있기 때문에, URL 에 VERSION 을 사용하면 안 됩니다. 파라미터 버전을 참조하세요.
- **identifier** (필수): 커버리지 이름
- **time** (선택적): 시점 (time position) 또는 기간 (time period) (beginPosition/endPosition[/timeResolution])
- **format** (선택적): 지원하는 포맷 이름입니다. 기본값은 이름에 `tif` 가 있는 경우 첫 번째로 지원하는 포맷 또는 그렇지 않은 경우 첫 번째로 지원하는 포맷입니다.
- **crs** (선택적) : 기관:ID 형식의, 예를 들면 EPSG:4326 같은 좌표계입니다. 기본값은 지원하는 경우 EPSG:4326, 지원하지 않는 경우 첫 번째로 지원하는 좌표계입니다.
- **username** (선택적): 기본 인증을 위한 사용자 이름입니다.
- **password** (선택적): 기본 인증을 위한 비밀번호입니다.
- **IgnoreGetMapUrl** (선택적, 꼼수): 지정한 (1 로 설정한) 경우, GetCapabilities 가 노출하는 GetCoverage URL 을 무시합니다. 서버를 제대로 환경설정하지 않은 경우 필요할 수도 있습니다.
- **InvertAxisOrientation** (선택적, 꼼수): 지정한 (1 로 설정한) 경우, GetCoverage 요청에 있는 축의 순서를 바꿉니다. 서버가 사용하는 축 순서가 틀린 경우 지리 좌표계에 대해 필요할 수도 있습니다.
- **IgnoreAxisOrientation** (선택적, 꼼수): 지정한 (1 로 설정한) 경우, 지리 좌표계에 대해 WCS 표준을 준수하는 축 순서로 바꾸지 않습니다.
- **cache** (선택적): QNetworkRequest::CacheLoadControl 에서 설명하는 대로 캐시 불러오기를 제어하지만, AlwaysCache 로 실패한 경우 요청을 PreferCache 로 다시 보냅니다. 다음 값들 가운데 하나로 설정할 수 있습니다: AlwaysCache, PreferCache, PreferNetwork, AlwaysNetwork. 기본값은 AlwaysCache 입니다.

아니면 WMS 서버에서 래스터 레이어를 불러올 수도 있습니다. 하지만 현재 API 에서 GetCapabilities 응답에 접근할 수가 없습니다—사용자가 원하는 레이어가 어떤 레이어인지 알고 있어야 합니다:

```
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')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

3.3 QgsProject 인스턴스

렌더링 작업에 열려져 있는 레이어를 사용하고 싶다면, `QgsProject` 클래스 인스턴스에 해당 레이어를 추가해야 한다는 사실을 기억하십시오. `QgsProject` 클래스 인스턴스는 레이어의 소유권을 가지며, 이후 응용 프로그램의 어느 부분에서든 레이어의 유일 ID 를 통해 해당 레이어에 접근할 수 있습니다. 프로젝트에서 레이어를 제거하는 경우, 유일 ID 도 삭제됩니다. 사용자는 QGIS 인터페이스에서 또는 `removeMapLayer()` 메소드를 사용하는 파이썬 코드를 통해 레이어를 제거할 수 있습니다.

`addMapLayer()` 메소드를 사용하면 현재 프로젝트에 레이어를 추가할 수 있습니다:

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

절대 위치에 레이어를 추가하려면:

```
# 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))
```

`removeMapLayer()` 메소드를 사용해서 레이어를 삭제하고 싶다면:

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

앞의 코드에서는 (레이어의 `id()` 메소드를 호출해서 얻을 수 있는) 레이어 ID 를 전달했지만, 레이어 객체 자체도 전달할 수 있습니다.

불러온 레이어와 레이어 ID 의 목록을 원한다면, `mapLayers()` 메소드를 사용하십시오:

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

CHAPTER 4

TOC(Table Of Contents) 에 접근하기

힌트

PyQGIS 콘솔을 사용하지 않는 경우 이 페이지에 있는 코드 조각들을 다음과 같이 가져와야 합니다:

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

서로 다른 클래스들을 사용해서 차례 (Table Of Contents) 에 불러온 모든 레이어에 접근할 수 있으며, 다음 클래스들을 통해 정보를 검색할 수 있습니다:

- `QgsProject`
- `QgsLayerTreeGroup`

4.1 QgsProject 클래스

`QgsProject` 클래스를 사용하면 차례 및 불러온 모든 레이어에 대한 정보를 검색할 수 있습니다.

불러온 레이어들을 가져오려면 `QgsProject` 클래스의 `instance` 를 생성하고 이 클래스의 메소드를 사용해야 합니다.

주 메소드는 `mapLayers()` 입니다. 이 메소드가 불러온 레이어들의 목록 (dictionary) 을 반환할 것입니다:

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

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

이 목록의 `keys` 는 유일한 레이어 ID 이고 `values` 는 관련 객체입니다.

이제 레이어에 대한 다른 모든 정보를 간단하게 얻을 수 있습니다:

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

print(layers_list)
```

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

레이어의 이름을 사용해서 차례도 쿼리할 수 있습니다:

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

> **참고**
>
> 이름이 일치하는 모든 레이어들의 목록을 반환하기 때문에, 이 이름을 가진 첫 번째 레이어를 가져오기 위해 `[0]` 인덱스를 사용합니다.

4.2 QgsLayerTreeGroup 클래스

레이어 트리는 노드 (node) 들로 구성된 전통적인 트리 구조입니다. 현재 두 가지 노드 유형이 존재합니다: 그룹 노드 (`QgsLayerTreeGroup`) 와 레이어 노드 (`QgsLayerTreeLayer`) 입니다.

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

`QgsProject` 클래스의 `layerTreeRoot()` 메소드를 사용하면 프로젝트 레이어 트리에 쉽게 접근할 수 있습니다:

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

`root` 는 그룹 노드이며 하위 *(children)* 노드를 가지고 있습니다:

```
root.children()
```

직계 하위 노드 목록을 반환합니다. 하위 그룹 노드는 그 직계 상위 노드로부터 접근해야 합니다.

하위 노드 가운데 하나를 검색할 수 있습니다:

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

```
<QgsLayerTreeLayer: countries>
```

레이어의 (유일) `id` 를 사용해서도 레이어를 검색할 수 있습니다:

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

그리고 그룹의 이름을 사용해서도 그룹을 검색할 수 있습니다:

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

`QgsLayerTreeGroup` 클래스는 차례에 대한 더 많은 정보를 가져오는 데 사용할 수 있는 유용한 다른 메소드들을 다수 가지고 있습니다:

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

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

이제 프로젝트의 레이어 트리에 레이어를 몇 개 추가해봅시다. 두 가지 방법이 있습니다:

1. **명시적 추가**—`addLayer()` 또는 `insertLayer()` 함수를 사용합니다:

```
# 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. **암시적 추가**—프로젝트의 레이어 트리가 레이어 레지스트리와 연결돼 있기 때문에 맵 레이어 레지스트리에 레이어를 추가해주는 것으로 충분합니다:

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

`QgsVectorLayer` 와 `QgsLayerTreeLayer` 클래스를 쉽게 전환할 수 있습니다:

```
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)>
```

`addGroup()` 메소드를 사용하면 그룹을 쉽게 추가할 수 있습니다. 다음 예시에서, 전자는 차례의 마지막에 그룹을 추가할 것이지만 후자의 경우 기존 그룹 안에 또다른 그룹을 추가할 수 있습니다:

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

노드 및 그룹을 이동시키고 싶다면, 유용한 메소드들이 많이 있습니다.

기존 노드의 이동은 세 단계로 이루어집니다:

1. 기존 노드 복제
2. 복제한 노드를 원하는 위치로 이동
3. 원본 노드 삭제

```
# 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)
```

범례에서 레이어를 이동시키는 작업은 좀 더 복잡합니다:

```
# 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)
```

또는 기존 그룹으로 이동시키는 작업도 그렇죠:

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

몇몇 다른 메소드들을 사용해서 그룹과 레이어를 수정할 수 있습니다:

```
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)
```

CHAPTER 5

래스터 레이어 사용하기

힌트

PyQGIS 콘솔을 사용하지 않는 경우 이 페이지에 있는 코드 조각들을 다음과 같이 가져와야 합니다:

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

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

5.1 레이어 상세 정보

래스터 레이어는 하나 또는 그 이상의 래스터 밴드로 이루어져 있습니다—이를 단일 밴드 그리고 다중 밴드 래스터라고 합니다. 밴드 하나는 값들의 행렬을 표현합니다. 컬러 이미지 (예: 항공 사진) 는 적색, 청색, 녹색 밴드로 이루어진 래스터입니다. 단일 밴드 래스터는 일반적으로 연속적인 변수들 (예: 표고) 또는 불연속적인 변수들 (예: 토지 이용도) 가운데 하나를 표현합니다. 또는 래스터 레이어에 색상표가 첨부되어 래스터 값들이 색상표에 저장된 색상들을 참조하는 경우도 있습니다.

다음 코드는 `rlayer` 가 `QgsRasterLayer` 클래스 객체라고 가정합니다.

```
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 렌더링 작업자

래스터 레이어를 불러왔을 때, 그 유형을 기반으로 기본 렌더링 작업자가 할당됩니다. 레이어 속성에서 또는 프로그래밍 방식으로 렌더랑 작업자를 변경할 수 있습니다.

현재 렌더링 작업자를 쿼리하려면:

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

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

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

```
singlebandgray
```

렌더링 작업자를 설정하려면 `QgsRasterLayer` 클래스의 `setRenderer()` 메소드를 사용하십시오. (`QgsRasterRenderer` 클래스에서 파생된) 렌더링 작업자 클래스가 여러 개 있습니다:

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

단일 밴드 래스터는 회색조 (낮은 값은 검정색, 높은 값은 흰색) 또는 값에 색상을 할당하는 의사 색상 (pseudo-color) 알고리즘 가운데 하나로 그릴 수 있습니다. 색상표를 가진 단일 밴드 래스터는 색상표를 사용해서도 그릴 수 있습니다. 다중 밴드 래스터는 일반적으로 RGB 색상에 밴드들을 매핑시켜서 그립니다. 또는 밴드 단 한 개만 사용해서 그릴 수도 있습니다.

5.2.1 단일 밴드 래스터

단일 밴드 래스터 레이어를 (0 에서 255 까지의 픽셀 값들에 대응하는) 초록색에서 노란색 범위의 색상을 사용해서 렌더링하려 한다고 해봅시다. 첫 번째 단계는 `QgsRasterShader` 클래스 객체를 준비해서 그 셰이더 (shader) 함수를 환경설정하는 것입니다:

```
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)
```

셰이더는 자체 색상표가 지정하는 대로 색상을 매핑합니다. 이 색상표가 픽셀 값들과 그 관련 색상들의 목록을 제공합니다. 세 가지 보간 모드가 있습니다:

- 선형 (`Interpolated`): 픽셀 값 위아래 있는 색상표 항목들로부터 색상을 선형적으로 보간합니다.
- 불연속 (`Discrete`): 픽셀 값 이상의 값을 가진 색상표 항목들 가운데 픽셀 값과 가장 가까이 있는 색상표 항목의 색상을 사용합니다.
- 정확 (`Exact`): 색상을 보간하지 않고, 색상표 항목들과 동등한 값을 가진 픽셀들만 그릴 것입니다.

두 번째 단계에서는 래스터 레이어에 이 셰이더를 할당할 것입니다:

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

앞의 코드에 있는 숫자 `1` 은 밴드 번호입니다. (래스터 밴드의 인덱스는 1 부터 시작합니다.)

마지막으로 결과물을 보려면 `triggerRepaint()` 메소드를 사용해야 합니다:

```
rlayer.triggerRepaint()
```

5.2.2 다중 밴드 래스터

QGIS 는 컬러 이미지를 생성하기 위해 기본적으로 처음 3 개의 밴드를 적색, 녹색, 청색으로 매핑합니다. (이를 `MultiBandColor` 그리기 스타일이라 합니다.) 그러나 이 설정을 무시하고 싶은 경우가 있을 수도 있습니다. 다음은 적색 밴드 (1) 와 녹색 밴드 (2) 를 교환하는 코드입니다:

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

래스터를 가시화하는 데 밴드 하나만 필요한 경우, 단일 밴드 그리기를 회색조 또는 의사 색상 가운데 하나로 선택하면 됩니다.

맵을 업데이트해서 결과물을 보려면 `triggerRepaint()` 메소드를 사용해야 합니다:

```
rlayer_multi.triggerRepaint()
```

5.3 값 쿼리하기

`QgsRasterDataProvider` 클래스의 `sample()` 메소드를 사용하면 래스터 값을 쿼리할 수 있습니다. `QgsPointXY` 클래스와 여러분이 쿼리하려는 래스터 레이어의 밴드 번호를 지정해야 합니다. 이 메소드는 래스터 값과 쿼리 결과에 따라 `True` 또는 `False` 를 가진 투플 (tuple) 을 반환합니다:

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

래스터 값을 쿼리할 수 있는 또다른 메소드는 `QgsRasterIdentifyResult` 클래스 객체를 반환하는 `identify()` 메소드입니다.

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

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

```
{1: 323.0}
```

이 경우, `results()` 메소드가 밴드 인덱스를 키로 그리고 밴드 값을 값으로 가지는 목록 (dictionary) 을 반환합니다. 예를 들면 `{1: 323.0}` 같은 식으로 말이죠.

5.4 래스터 데이터 편집하기

`QgsRasterBlock` 클래스를 사용하면 래스터 레이어를 생성할 수 있습니다. 예를 들어 픽셀 당 1 바이트를 가진 2x2 래스터 블록을 생성하려면:

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

`writeBlock()` 메소드 덕분에 래스터 픽셀을 덮어쓸 수 있습니다. 2x2 블록의 0,0 위치에 있는 기존 래스터 데이터를 덮어쓰려면:

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

CHAPTER 6

벡터 레이어 사용하기

힌트

PyQGIS 콘솔을 사용하지 않는 경우 이 페이지에 있는 코드 조각들을 다음과 같이 가져와야 합니다:

```
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,
)
```

이 장에서는 벡터 레이어에 대해 할 수 있는 여러 가지 작업들을 소개합니다.

이 장에서 소개하는 대부분의 작업은 `QgsVectorLayer` 클래스의 메소드들에 바탕을 두고 있습니다.

6.1 속성에 대한 정보 검색하기

`QgsVectorLayer` 클래스 객체에 대해 `fields()` 메소드를 호출하면 벡터 레이어와 연관된 필드들에 대한 정보를 검색할 수 있습니다:

```
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()` 와 `mapTipTemplate()` 메소드는 maptips 탭에서 쓰이는 필드와 템플릿에 대한 정보를 제공합니다.

벡터 레이어를 불러왔을 때 QGIS 는 언제나 필드 하나를 `Display Name` 으로 선택하는 반면, `HTML Map Tip` 은 기본적으로 비어 있습니다. 이 메소드들을 사용하면 둘 다 손쉽게 가져올 수 있습니다:

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

```
name
```

> **참고**
>
> `Display Name` 을 필드에서 표현식으로 변경할 경우, `displayField()` 대신 `displayExpression()` 메소드를 사용해야 합니다.

6.2 벡터 레이어 작업 반복하기

벡터 레이어에 있는 피처들에 대한 작업을 반복하는 일은 가장 흔한 작업 가운데 하나입니다. 다음은 이런 작업을 수행하는 단순한 기본 코드의 예시로, 각 피처에 대한 몇몇 정보를 출력합니다. `layer` 변수가 `QgsVectorLayer` 클래스를 가지고 있다고 가정합니다.

```
# "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
    break
```

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

6.3 피처 선택하기

QGIS 데스크탑에서 피처를 여러 가지 방법으로 선택할 수 있습니다: 여러분은 피처 하나를 클릭하거나, 맵 캔버스 위에 직사각형을 그리거나, 또는 표현식 필터를 사용할 수 있습니다. 선택 집합에 여러분의 주의를 끌기 위해, 선택된 피처는 보통 다른 (기본값은 노란색) 색상으로 강조됩니다.

때로는 프로그래밍 방식으로 피처를 선택하거나 또는 기본 색상을 변경하는 편이 유용할 수도 있습니다.

피처를 모두 선택하려면, `selectAll()` 메소드를 사용하면 됩니다:

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

표현식을 사용해서 선택하려면, `selectByExpression()` 메소드를 사용하십시오:

```
# 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)
```

선택 집합의 색상을 변경하려면 다음 예시에서 볼 수 있는 바와 같이 `QgsMapCanvas` 클래스의 `setSelectionColor()` 메소드를 사용하면 됩니다:

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

지정한 레이어의 선택 피처 목록에 피처를 추가하려면, 피처 ID 목록에 피처 ID 를 전달하는 `select()` 메소드를 호출하면 됩니다:

```
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)
```

선택 집합을 지우려면:

```
layer.removeSelection()
```

6.3.1 속성에 접근하기

속성은 속성 이름으로 참조할 수 있습니다:

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

```
First feature
```

아니면, 인덱스로 속성을 참조할 수 있습니다. 이름을 사용하는 방법보다 좀 더 빠릅니다. 예를 들어 두 번째 속성을 가져오려면:

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

```
First feature
```

6.3.2 선택한 피처에 대한 작업 반복하기

선택한 피처들만 필요한 경우, 벡터 레이어에서 `selectedFeatures()` 메소드를 사용하면 됩니다:

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

6.3.3 피처 부분 집합에 대한 작업 반복하기

레이어에 있는 피처들 가운데 지정한 부분 집합에만, 예를 들면 지정한 영역 안에 있는 피처에만 작업을 반복하고 싶은 경우, `getFeatures()` 메소드 호출에 `QgsFeatureRequest` 클래스 객체를 추가해야 합니다. 다음은 그 예시입니다:

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

request = QgsFeatureRequest().setFilterRect(areaOfInterest)

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

속도를 향상시키기 위해, 교차 여부를 확인하는 데 피처의 경계 상자만 사용하는 경우가 많습니다. 하지만 교차하는 피처들만 반환되도록 보장하는 `ExactIntersect` 플래그가 존재합니다:

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

`setLimit()` 메소드를 사용하면 요청한 피처들의 개수를 제한할 수 있습니다. 다음은 그 예시입니다:

```
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>
```

앞의 예시에서 볼 수 있는 것과 같은 공간 필터 대신 (또는 추가로) 속성 기반 필터가 필요한 경우, `QgsExpression` 클래스 객체를 작성해서 `QgsFeatureRequest` 작성자 (constructor) 클래스로 전달하면 됩니다. 다음은 그 예시입니다:

```
# 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 클래스가 지원하는 문법 (syntax) 에 대해 자세히 알고 싶다면값을 필터링하고 계산하는 표현식을 참조하세요.

요청을 각 피처에 대해 검색된 데이터를 정의하기 위해 사용할 수 있기 때문에, 반복 작업자 (iterator) 가 모든 피처를 반환하긴 하지만 각 피처의 부분적인 데이터를 반환합니다.

```
# 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 벡터 레이어 수정하기

벡터 데이터 제공자 대부분은 레이어 데이터의 편집을 지원합니다. 가능한 편집 작업의 일부만 지원하는 경우도 있습니다. 어떤 기능들을 지원하는지 알고 싶다면 capabilities() 함수를 사용해보세요.

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

```
The layer supports DeleteFeatures
```

사용할 수 있는 케이퍼빌리티의 전체 목록을 보고 싶다면 QgsVectorDataProvider 의 API 문서 클래스를 참조해주십시오.

레이어의 케이퍼빌리티를 쉼표로 구분된 설명 텍스트로 출력하려면 다음 예시처럼 capabilitiesString() 메소드를 사용하면 됩니다:

```
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'
```

벡터 레이어 편집을 위한 다음 방법들 가운데 어떤 방법을 사용하든, 변경 사항은 기저 데이터 저장소 (파일, 데이터베이스 등등) 에 직접 반영됩니다. 일시적으로만 변경하고자 할 경우, 편집 버퍼로 수정 하는 방법을 설명하는 다음 절로 넘어가십시오.

참고

QGIS 안에서 (콘솔에서든 플러그인에서든) 작업하는 경우, 여러분이 도형의 스타일이나 속성을 변경한 결과를 보기 위해 맵 캔버스를 강제로 다시 그리게 해야 할 필요가 있을 수도 있습니다:

```
# 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 피처 추가하기

`QgsFeature` 클래스 인스턴스를 몇 개 생성한 다음 제공자의 `QgsVectorDataProvider` `addFeatures()` 메소드에 그 목록을 전달하십시오. 이 메소드가 결과 (`True` 또는 `False`) 와 추가된 피처 목록 (데이터 저장소가 설정한, 추가된 피처의 피처 ID) 이라는 값 2 개를 반환할 것입니다.

피처 속성을 설정하려면, (벡터 레이어의 `fields()` 메소드에서 얻을 수 있는) `QgsFields` 클래스 객체를 전달하는 피처를 초기화하거나, 또는 여러분이 추가하고 싶은 필드들의 개수를 전달하는 `initAttributes()` 메소드를 호출하면 됩니다.

```
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 피처 삭제하기

피처를 몇 개 삭제하려면, 삭제하려는 피처의 ID 목록만 지정하면 됩니다.

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

6.4.3 피처 수정하기

피처의 도형을 변경하거나 아니면 속성을 몇 개 변경할 수 있습니다. 다음 예시에서는 먼저 인덱스가 0 과 1 인 속성들의 값을 변경한 다음, 피처의 도형을 변경합니다.

```
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 })
```

> **팁**
>
> **도형만 편집하는 경우 QgsVectorLayerEditUtils 클래스 권장**
>
> 도형만 변경해야 하는 경우, 도형을 편집하는 몇 가지 유용한 (꼭짓점 변위, 삽입, 이동 등등의) 메소드들을 제공하는 `QgsVectorLayerEditUtils` 클래스를 사용하는 편이 좋을 수도 있습니다.

6.4.4 편집 버퍼로 벡터 레이어 수정하기

QGIS 응용 프로그램 안에서 벡터를 편집할 때, 먼저 특정 레이어에 대한 편집 모드를 시작한 다음 어떤 수정 작업을 하고 마지막으로 변경 사항을 커밋 (또는 롤백) 해야 합니다. 여러분이 무엇을 수정했든, 모든 변경 사항은 커밋하기 전까지는 작성되지 않습니다—레이어의 인메모리 (in-memory) 편집 버퍼에 머물 뿐입니다. 이 기능을 프로그래밍 방식으로도 사용할 수 있습니다—그저 데이터 제공자를 직접 사용하는 작업을 보완하는 벡터 레이어 편집의 또다른 방법일 뿐입니다. 벡터 레이어 편집을 위한 몇몇 GUI 도구들을 제공할 때 이 옵션을 사용하십시오. 여러분이 커밋/롤백할지 여부를 결정할 수 있게 해주고, 실행 취소 (undo)/다시 실행 (redo) 기능을 사용할 수 있게 해주기 때문입니다. 변경 사항을 커밋하면, 데이터 제공자에 편집 버퍼에 있는 변경 사항을 전부 저장합니다.

이 메소드들은 제공자에서 볼 수 있는 메소드들과 비슷하지만, 제공자에서와는 달리 `QgsVectorLayer` 클래스 객체에 대해 호출됩니다.

이 메소드들이 작동하기 위해서는 레이어가 반드시 편집 모드 상태여야만 합니다. 편집 모드를 시작하려면, `startEditing()` 메소드를 사용하십시오. 편집을 끝내려면, `commitChanges()` 또는 `rollBack()` 메소드를 사용하십시오. 전자는 데이터 소스에 여러분의 변경 사항을 전부 저장할 것이고, 후자는 변경 사항을 버리고 데이터 소스를 하나도 수정하지 않을 것입니다.

레이어가 편집 모드 상태인지 알아내려면, `isEditable()` 메소드를 사용하십시오.

다음은 이 편집 메소드들을 사용하는 방법을 보여주는 몇몇 예시입니다.

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

실행 취소/다시 실행이 제대로 작동하게 하려면, 앞에서 설명한 호출을 실행 취소 명령어 안에 래핑 (wrapping) 시켜야 합니다. (실행 취소/다시 실행에는 관심이 없고 변경하는 즉시 저장되기를 바라는 경우, 데이터 제공자로 편집하기 를 통해 더 쉽게 작업하게 될 것입니다.)

다음은 실행 취소 기능을 어떻게 사용할 수 있는지 보여주는 예시입니다:

```
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()` 메소드가 내부"활성"명령어를 생성한 다음 벡터 레이어의 순차적인 변경 사항들을 기록할 것입니다. `endEditCommand()` 메소드를 호출하면 명령어를 실행 취소 스택으로 넘겨서 여러분이 GUI 에서 실행 취소/다시 실행할 수 있게 될 것입니다. 벡터를 변경하는 도중 무언가 문제가 발생한 경우, `destroyEditCommand()` 메소드가 명령어를 제거하고 해당 명령어가 활성 상태인 동안 발생한 모든 변경 사항들을 롤백시킬 것입니다.

또한 다음 예시에서 볼 수 있는 바와 같이 `with edit(layer)` 선언문을 사용해서 커밋과 롤백을 좀 더 의미를 알 수 있는 (sementic) 코드 블록으로 래핑시킬 수 있습니다:

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

이 코드는 마지막에 `commitChanges()` 메소드를 자동 호출할 것입니다. 어떤 예외가 발생하는 경우, 이 코드는 모든 변경 사항을 롤백시키는 `rollBack()` 메소드를 호출할 것입니다. `commitChanges()` 메소드가 실행되는 동안 문제가 발생하는 경우 (이 메소드가 거짓을 반환하는 경우) `QgsEditError` 예외가 발생할 것입니다.

6.4.5 필드 추가하기 및 제거하기

필드 (속성) 를 추가하려면, 필드를 정의한 리스트를 지정해야 합니다. 필드를 삭제하려면 필드 인덱스 목록만 넘겨주면 됩니다.

```
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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

# 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)
```

데이터 제공자에서 필드를 추가 또는 제거한 다음 레이어의 필드를 업데이트해야 합니다. 변경 사항이 자동적으로 반영되지 않기 때문입니다.

```
layer.updateFields()
```

팁

`with` **기반 명령어를 사용해서 변경 사항을 직접 저장하기**

`with edit(layer):` 선언문을 사용하면 마지막에 `commitChanges()` 메소드를 자동으로 호출해서 변경 사항을 커밋시킬 것입니다. 어떤 예외가 발생하는 경우, `rollBack()` 메소드를 호출해서 모든 변경 사항을 롤백시킬 것입니다. 편집 버퍼로 벡터 레이어 수정하기 를 참조하세요.

6.5 공간 인덱스 사용하기

벡터 레이어에 대해 자주 쿼리를 해야 할 경우 공간 인덱스를 사용하면 코드 실행 속도를 획기적으로 향상시킬 수 있습니다. 여러분이 보간 알고리즘을 작성하는데, 보간 값을 계산하기 위해 지정한 위치에서 가장 가까운 포인트 레이어의 포인트 10 개를 알아내야 한다고 한번 상상해보십시오. 공간 인덱스가 없다면, QGIS 가 그 포인트 10 개를 찾는 유일한 방법은 해당 위치에서 모든 포인트까지의 거리를 각각 계산한 다음 그 거리들을 비교하는 것입니다. 특히 이 작업을 몇 군데의 위치에 대해 반복해야 할 경우 시간이 아주 오래 걸릴 수 있습니다. 레이어가 공간 인덱스를 가지고 있다면, 훨씬 효율적으로 작업할 수 있습니다.

공간 인덱스가 없는 레이어를 전화번호가 정렬되지도 색인되지도 않은 전화번호부라고 생각해보십시오. 어떤 사람의 전화번호를 찾으려면 처음부터 그 번호를 찾을 때까지 읽을 수밖에 없습니다.

QGIS 벡터 레이어에 공간 인덱스를 기본으로 생성하지는 않지만, 쉽게 생성할 수 있습니다. 여러분이 해야 할 일은 다음과 같습니다:

- `QgsSpatialIndex` 클래스를 사용해서 공간 인덱스를 생성합니다:

  ```
  index = QgsSpatialIndex()
  ```

- 인덱스에 피처를 추가합니다—인덱스는 `QgsFeature` 클래스 객체를 받아 내부 데이터 구조에 추가합니다. 여러분이 객체를 직접 생성할 수도 있고, 또는 제공자의 `getFeatures()` 메소드를 호출했을 때 받은 객체를 사용할 수도 있습니다.

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

- 아니면 벌크 불러오기를 사용해서 레이어의 피처들을 한번에 전부 불러올 수 있습니다.

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

- 공간 인덱스가 값들로 채워지고 나면 쿼리를 해볼 수 있습니다.

```
# 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))
```

`QgsSpatialIndexKDBush` 클래스 공간 인덱스도 사용할 수 있습니다. 이 인덱스는 표준 `QgsSpatialIndex` 클래스 인덱스와 비슷하지만:

- 단일 포인트 피처 **만** 지원합니다.
- **정적** 입니다. (인덱스 작성 후에 인덱스에 어떤 피처도 추가할 수 없습니다.)
- **훨씬 빠릅니다!**
- 추가적인 피처 요청 없이도 원본 피처의 포인트들을 직접 검색할 수 있습니다.
- 예를 들어 검색 포인트로부터 어떤 반경 안에 들어오는 모든 포인트를 반환받는다거나 하는, 진정한 거리 기반 검색을 지원합니다.

6.6 QgsVectorLayerUtils 클래스

`QgsVectorLayerUtils` 클래스는 벡터 레이어에 사용할 수 있는 몇몇 매우 유용한 메소드들을 담고 있습니다.

예를 들면 `createFeature()` 메소드는 벡터 레이어에 추가될, 레이어의 각 필드의 모든 최종 제약조건과 기본값을 유지하는 `QgsFeature` 클래스를 준비합니다:

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

`getValues()` 메소드는 필드 또는 표현식의 값을 빠르게 얻을 수 있게 해줍니다:

```
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 벡터 레이어 생성하기

몇 가지 방법으로 벡터 레이어 데이터셋을 생성할 수 있습니다:

- `QgsVectorFileWriter` 클래스: 디스크에 벡터 파일을 작성할 수 있는 편리한 클래스입니다. 전체 벡터 레이어를 저장하는 `writeAsVectorFormatV3()` 메소드를 정적 호출하거나, 클래스의 인스턴스를 생성한 다음 상속한 `addFeature()` 메소드를 호출합니다. 이 클래스는 GDAL 이 지원하는 벡터 포맷들을 모두 (GeoPackage, 셰이프파일, GeoJSON, KML 및 기타 등등) 지원합니다.
- `QgsVectorLayer` 클래스: 데이터에 연결하고 접근하기 위한 데이터 소스의 지정 경로 (URL) 을 해석하는 데이터 제공자를 인스턴스화합니다. 이 클래스를 사용하면 메모리 기반 임시 레이어 (`memory`) 를 생성하고 GDAL 벡터 데이터셋 (`ogr`), 데이터베이스 (`postgres`, `spatialite`, `mysql`, `mssql`), 그리고 기타 등등 (`wfs`, `gpx`, `delimitedtext`, …) 에 연결할 수 있습니다.

6.7.1 QgsVectorFileWriter 클래스의 인스턴스로부터

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

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

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

또한 `FieldValueConverter` 클래스를 사용하면 필드들을 서로 다른 포맷들과 호환되도록 변환할 수 있습니다. 예를 들어 (PostgreSQL 에서) 배열 변수 유형을 텍스트 유형으로 변환하려면 다음과 같이 하면 됩니다:

```
LIST_FIELD_NAME = 'xxxx'

class ESRIValueConverter(QgsVectorFileWriter.FieldValueConverter):

  def __init__(self, layer, list_field):
    QgsVectorFileWriter.FieldValueConverter.__init__(self)
    self.layer = layer
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    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
```

대상 좌표를 지정할 수도 있습니다—`QgsCoordinateReferenceSystem` 클래스의 무결한 인스턴스를 네 번째 파라미터로 전달하면, 레이어가 해당 좌표계로 변환됩니다.

무결한 드라이버 이름을 알고 싶다면 `supportedFiltersAndFormats()` 메소드를 호출하거나 supported formats by OGR (OGR 가 지원하는 포맷들) 문서를 살펴보십시오—"Code"열에 이 값을 드라이어 이름으로 전달해야 할 것입니다.

선택한 피처들만 내보낼지, 생성 용 드라이버 전용 심화 옵션들을 전달할지, 작성기에 속성을 생성하지 말라고 할지 등등의 여부를 선택적으로 설정할 수 있습니다. 기타 (선택적) 파라미터들이 여럿 있습니다. 자세한 내용은 `QgsVectorFileWriter` 클래스 문서를 참조하세요.

6.7.2 피처로부터 직접

```
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()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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)

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

6.7.3 `QgsVectorLayer` 클래스 인스턴스로부터

`QgsVectorLayer` 클래스가 지원하는 모든 데이터 제공자들 가운데, 메모리 기반 레이어를 주목해봅시다. 메모리 제공자는 주로 플러그인이나 제 3 자 응용 프로그램 개발자들이 사용할 목적으로 설계되었습니다. 메모리 제공자는 디스크 상에 데이터를 저장하지 않기 때문에, 개발자들이 몇몇 임시 레이어에 대한 빠른 백엔드로 사용할 수 있기 때문입니다.

이 제공자는 문자열 (string), 정수 (int), 더블형 실수 (double) 유형의 필드를 지원합니다.

메모리 제공자는 공간 인덱스 작업도 지원하는데, 제공자의 `createSpatialIndex()` 함수를 호출하면 됩니다. 공간 인덱스를 생성하고 나면 좁은 지역 안에 있는 피처들을 더 빠르게 반복 작업할 수 있게 될 것입니다. (피처들을 전부 처리할 필요 없이 지정한 직사각형 안에 있는 피처들만 처리하면 되기 때문입니다.)

`QgsVectorLayer` 클래스 작성자 (constructor) 에 `"memory"` 를 제공자 문자열로 전달하면 메모리 제공자가 생성됩니다.

이 작성자는 레이어의 도형 유형을 정의하는 URI 도 받습니다. 도형 유형은 다음 가운데 하나일 수 있습니다: `"Point"`, `"LineString"`, `"Polygon"`, `"MultiPoint"`, `"MultiLineString"`, `"MultiPolygon"` or `"None"`

좌표계, 필드, 메모리 기반 제공자의 인덱스 작업도 이 URI 로 설정할 수 있습니다. 문법은 다음과 같습니다:

crs=definition
: 좌표계를 지정합니다. 이때 좌표계 정의는 `QgsCoordinateReferenceSystem.createFromString()` 메소드가 받아들이는 어떤 양식도 될 수 있습니다.

index=yes
: 제공자가 공간 인덱스를 사용하도록 설정합니다.

field=name:type(length,precision)
레이어의 속성을 설정합니다. 속성의 명칭은 필수적이며, 선택적으로 유형 (정수, 더블형 실수, 문자 스트링), 길이 및 정밀도를 설정할 수 있습니다. 여러 개의 필드를 정의할 수도 있습니다.

다음은 이 모든 옵션들을 포함하는 URI 의 예시입니다.

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

다음은 메모리 기반 제공자를 생성하고 값을 채우는 코드의 예시입니다.

```
from qgis.PyQt.QtCore import QDateTime, QMetaType, Qt

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

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

# add a feature
fet = QgsFeature()
fet.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(10,10)))
fmt = Qt.DateFormat.ISODate
t = QDateTime.fromString("2000-01-01T12:00:00", fmt)
fet.setAttributes(["Johnny", 2, 0.3, t])
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()
```

마지막으로, 모든 작업이 성공적이었는지 확인해봅시다.

```
# 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: 4
features: 1
extent: 10.0 10.0 10.0 10.0
F: 1 ['Johnny', 2, 0.3, PyQt5.QtCore.QDateTime(2000, 1, 1, 12, 0)] <QgsPointXY:␣
↪POINT(10 10)>
```

6.8 벡터 레이어의 표현 (심볼)

벡터 레이어를 렌더링할 때, 레이어와 관련된 **렌더링 작업자** 와 **심볼** 이 데이터의 표현을 결정합니다. 심볼은 피처의 시각적 표현을 그리는 일을 담당하는 클래스이며, 렌더링 작업자는 특정 피처에 대해 어떤 심볼을 적용할지 결정합니다.

지정한 레이어의 렌더링 작업자는 다음과 같이 얻을 수 있습니다:

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

그리고 이를 참조해서 조금 더 나가봅시다.

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

```
Type: singleSymbol
```

QGIS 핵심 라이브러리에서 알려진 몇몇 렌더링 작업자 유형을 사용할 수 있습니다:

유형	클래스	설명
singleSymbol	QgsSingleSymbolRenderer	모든 피처를 동일한 심볼로 렌더링합니다.
categorizedSymbol	QgsCategorizedSymbolRenderer	피처를 각 카테고리 별로 서로 다른 심볼로 렌더링합니다.
graduatedSymbol	QgsGraduatedSymbolRenderer	피처를 값의 범위에 따라 서로 다른 심볼로 렌더링합니다.

몇몇 사용자 정의 렌더링 작업자 유형이 있을 수도 있기 때문에, 이 유형들만 있을 거라고 절대로 가정하지 마십시오. 현재 사용할 수 있는 렌더링 작업자들을 알아내려면 응용 프로그램의 QgsRendererRegistry 클래스를 쿼리하면 됩니다:

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

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

렌더링 작업자의 콘텐츠를 텍스트 형식으로 덤프받을 수도 있습니다—디버깅 작업에 유용할 수 있습니다.

```
renderer.dump()
```

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

6.8.1 단일 심볼 렌더링 작업자

symbol() 메소드를 호출하면 렌더링에 쓰인 심볼을 가져올 수 있고, setSymbol() 메소드로 심볼을 변경할 수 있습니다. (C++ 개발자는 기억하세요: 렌더링 작업자가 심볼을 소유합니다.)

setSymbol() 메소드를 호출해서 적당한 심볼의 인스턴스를 전달하면 특정 벡터 레이어가 사용하는 심볼을 변경할 수 있습니다. QgsMarkerSymbol, QgsLineSymbol, 그리고 QgsFillSymbol 클래스의 각각 대응하는 createSimple() 함수를 호출하면 각각 포인트, 라인, 폴리곤 레이어 용 심볼을 생성할 수 있습니다.

createSimple() 메소드에 전달되는 목록 (dictionary) 이 심볼의 스타일 속성을 설정합니다.

예를 들면 다음 코드 예시와 같이 `setSymbol()` 메소드를 호출해서 `QgsMarkerSymbol` 클래스의 인스턴스를 전달하면, 특정 **포인트** 레이어가 사용하는 심볼을 대체시킬 수 있습니다.

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

`name` 은 마커의 형태를 의미하며, 다음 중 어느 것이라도 가능합니다.

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

심볼 인스턴스의 첫 번째 심볼 레이어에 대한 전체 속성 목록을 알고 싶다면 다음 코드 예시를 따라해보십시오:

```
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'}
```

여러분이 속성을 일부 변경하고 싶다면 다음이 유용할 것입니다:

```
# 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 카테고리 심볼 렌더링 작업자

카테고리 렌더링 작업자를 사용할 때, 범주화에 사용되는 속성을 쿼리하고 설정할 수 있습니다: `classAttribute()` 및 `setClassAttribute()` 메소드를 사용하십시오.

카테고리 목록을 얻는 방법은 다음과 같습니다.

```
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>
```

이때 `value()` 메소드가 카테고리들을 구분하는 데 쓰이는 값이고, `label()` 메소드는 카테고리 설명에 쓰이는 텍스트이며, `symbol()` 메소드는 할당된 심볼을 반환합니다.

이 렌더링 작업자는 일반적으로 범주화에 사용된 원본 심볼과 색상표도 저장합니다: `sourceColorRamp()` 및 `sourceSymbol()` 메소드를 사용하십시오.

6.8.3 등급 심볼 렌더링 작업자

이 렌더링 작업자는 앞에서 설명한 카테고리 심볼 렌더링 작업자와 매우 비슷하지만, 범주 당 하나의 속성을 할당하는 대신 값의 범위에 따라 할당하기 때문에 숫자 속성에만 사용할 수 있습니다.

이 렌더링 작업자가 사용하는 범위에 대해 상세히 알아내려면 다음과 같은 방법을 사용합니다.

```
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>
```

(범주화 속성 이름을 알아내기 위한) `classAttribute()` 메소드와 `sourceSymbol()` 및 `sourceColorRamp()` 메소드를 다시 사용하면 됩니다. 범위 생성 방법을 결정하는 추가적인 `mode()` 메소드도 있습니다: 등간격을 사용할지, 사분위를 사용할지, 또는 기타 다른 방법을 사용할지 결정할 수 있습니다.

사용자 정의 등급 심볼 렌더링 작업자를 생성하고자 할 경우 다음 예시에서처럼 하면 됩니다. (이 예시에서는 범주 2 개인 간단한 배열을 생성합니다.)

```
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'
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 심볼 다루기

심볼을 표현하기 위한 `QgsSymbol` 기저 (base) 클래스는 파생 클래스 3 개를 가지고 있습니다:

- `QgsMarkerSymbol` —포인트 피처 용
- `QgsLineSymbol` —라인 피처 용
- `QgsFillSymbol` —폴리곤 피처 용

모든 심볼은 하나 이상의 (`QgsSymbolLayer` 클래스에서 파생된 클래스인) **심볼 레이어로 이루어집니다**. 이 심볼 레이어들이 실제 렌더링 작업을 하며, 심볼 클래스 자체는 심볼 레이어 용 컨테이너 역할만 합니다.

(예를 들어 렌더링 작업자로부터 나온) 심볼 인스턴스를 가지고 있다면, 심볼을 탐색해볼 수 있습니다: `type()` 메소드는 심볼이 마커, 라인, 또는 채우기 심볼 가운데 어떤 유형인지 알려줍니다. 심볼의 짧은 설명을 반환하는 `dump()` 메소드도 있습니다. 심볼 레이어 목록을 보고 싶다면:

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

```
0: SimpleMarker
```

심볼의 색상을 알아내려면 `color()` 메소드를 사용하고 그 색상을 변경하려면 `setColor()` 메소드를 사용하십시오. 마커 심볼의 경우 추가적으로 `size()` 와 `angle()` 메소드를 사용해서 각각 심볼 크기와 회전 각도를 쿼리할 수 있습니다. 라인 심볼의 경우 `width()` 메소드가 라인 너비를 반환합니다.

기본적으로 크기 및 너비는 밀리미터 단위이며, 각도는 도 단위입니다.

심볼 레이어 다루기

앞에서 말했듯이, (`QgsSymbolLayer` 클래스의 하위 클래스인) 심볼 레이어가 피처의 모양을 결정합니다. 일반적인 사용례를 위한 기본 심볼 레이어 클래스가 몇 개 있습니다. 새로운 심볼 레이어 유형을 구현할 수 있기 때문에, 사용자가 피처를 어떻게 렌더링할지를 임의로 정의할 수 있습니다. `layerType()` 메소드가 심볼 레이어 클래스를 고유하게 식별합니다—기본 클래스는 `SimpleMarker`, `SimpleLine` 및 `SimpleFill` 심볼 레이어 유형입니다.

지정한 심볼 레이어 클래스에 생성할 수 있는 심볼 레이어 유형들의 전체 리스트를 다음 코드로 볼 수 있습니다:

```
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` 클래스는 사용할 수 있는 모든 레이어 유형들의 데이터베이스를 관리합니다.

심볼 레이어 데이터에 접근하려면, 심볼 모양을 결정하는 속성들의 키-값 목록 (dictionary) 을 반환하는 `properties()` 메소드를 사용하십시오. 각각의 심볼 레이어 유형은 해당 유형이 사용하는 특정한 속성 집합을 가지고 있습니다. 또한 각각의 설정자 (setter) 대응 항목을 가진 일반적인 `color()`, `size()`, `angle()` 및 `width()` 메소드들도 있습니다. 물론 크기와 각도는 마커 심볼에만 사용할 수 있으며 너비는 라인 심볼에만 사용할 수 있습니다.

사용자 지정 심볼 레이어 유형 생성하기

데이터를 어떻게 렌더링할지 사용자 정의하고 싶다고 상상해보십시오. 사용자가 원하는 방식대로 피처를 그리는, 자신만의 심볼 레이어 클래스를 생성할 수 있습니다. 다음 코드는 지정된 반경으로 빨간색 원을 그리는 마커의 예시입니다:

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

class FooSymbolLayer(QgsMarkerSymbolLayer):

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  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()` 메소드가 심볼 레이어의 이름을 결정합니다. 이 이름은 전체 심볼 레이어들 사이에서 유일해야 합니다. `properties()` 메소드는 속성들의 지속성을 위해 쓰입니다. `clone()` 메소드는 반드시 심볼 레이어와 모든 속성이 정확하게 동일한 복제물을 반환해야만 합니다. 마지막으로 렌더링 메소드들이 있습니다: 첫 번째 피처를 렌더링하기 전에 `startRender()` 메소드를 호출하고, 렌더링을 완료했을 때 `stopRender()` 메소드를 호출하며, 렌더링을 하기 위해서는 `renderPoint()` 메소드를 호출합니다. 포인트 (들) 의 좌표는 이미 산출 좌표로 변환되어 있습니다.

폴리라인 및 폴리곤의 경우 유일한 차이점은 렌더링 메소드에 있을 것입니다: 라인의 경우 라인 목록을 받는 `renderPolyline()` 메소드를 사용하는 반면, 폴리곤의 경우 외곽 고리 상에 있는 포인트 목록을 첫 번째 파라미터로 그리고 내곽 고리의 포인트 목록 (또는 None) 을 두 번째 파라미터로 받는 `renderPolygon()` 메소드를 사용할 것입니다.

일반적으로 사용자가 모양을 변경할 수 있도록 심볼 레이어 유형의 속성을 설정하기 위한 GUI 를 추가하는 편이 좋습니다. 앞의 예시에서 사용자가 원의 반경을 설정하도록 할 수 있습니다. 다음 코드는 그런 위젯을 구현하는 예시입니다:

```
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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        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()"))
```

심볼 속성 대화창 안에 이 위젯을 내장시킬 수 있습니다. 심볼 속성 대화창에서 심볼 레이어 유형을 선택했을 때, 심볼 레이어의 인스턴스 및 심볼 레이어 위젯의 인스턴스를 생성합니다. 그 다음 `setSymbolLayer()` 메소드를 호출해서 위젯에 심볼 레이어를 할당합니다. 이때 위젯이 심볼 레이어의 속성을 반영하도록 UI 를 업데이트해야 합니다. 심볼 속성 대화창에서 심볼에 사용할 수 있도록 심볼 레이어를 다시 검색하는 데 `symbolLayer()` 메소드를 사용합니다.

속성을 변경할 때마다, 이 위젯은 심볼 속성 대화창이 심볼 미리보기를 업데이트할 수 있도록 `changed()` 신호를 보내야 합니다.

이제 마지막 단계만 남았습니다. QGIS 가 이 새 클래스들을 인식하도록 심볼 레이어를 레지스트리에 추가하는 일입니다. 심볼 레이어를 레지스트리에 추가하지 않고 사용할 수도 있지만, 예를 들어 사용자 정의 심볼 레이어를 담고 있는 프로젝트 파일을 불러오지 못한다든가 GUI 에서 레이어 속성을 편집할 수 없다는 등 몇몇 기능들을 사용할 수 없게 됩니다.

심볼 레이어에 대한 메타데이터도 생성해야 합니다:

```
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)
```

상위 클래스의 작성자에 (레이어가 반환하는 유형과 동일한) 레이어 유형과 심볼 유형 (마커/라인/채우기) 을 전달해야 합니다. `createSymbolLayer()` 메소드가 `props` 목록 (dictionary) 에 지정된 속성들을 가진 심볼 레이어의 인스턴스 생성을 처리합니다. 그리고 해당 심볼 레이어 유형을 위한 설정 위젯을 반환하는 `createSymbolLayerWidget()` 메소드도 있습니다.

마지막 단계는 레지스트리에 이 심볼 레이어를 추가하는 일입니다. 이제 모두 끝났습니다.

6.8.5 사용자 정의 렌더링 작업자 생성하기

피처를 렌더링하는 데 어떻게 심볼을 선택할지에 대한 규칙을 마음대로 지정하고 싶을 경우 새로운 렌더링 작업자를 만드는 편이 유용할 수도 있습니다. 여러 필드들을 조합해서 심볼을 결정해야 하거나, 현재 축척에 따라 심볼 크기를 변경해야 하는 등의 경우에 새로운 렌더링 작업자를 만들면 좋습니다.

다음 코드는 마커 심볼 2 개를 생성해서 각 피처마다 랜덤하게 2 개 중 1 개를 할당하는 간단한 사용자 정의 렌더링 작업자의 예시입니다:

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


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

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

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

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

  def usedAttributes(self, context):
    return []

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

상위 `QgsFeatureRenderer` 클래스의 작성자는 (렌더링 작업자들 가운데 유일해야 하는) 렌더링 작업자 이름을 필요로 합니다. `symbolForFeature()` 가 특정 피처에 어떤 심볼을 사용할지 결정하는 메소드입니다. `startRender()` 와 `stopRender()` 메소드는 각각 심볼 렌더링 작업의 시작과 마무리를 처리합니다. `usedAttributes()` 메소드는 렌더링 작업자가 존재할 것으로 예상하는 필드 이름 목록을 반환할 수 있습니다. 마지막으로 `clone()` 메소드는 렌더링 작업자의 복제물을 반환할 것입니다.

심볼 레이어와 마찬가지로, 렌더링 작업자를 환경설정하기 위한 GUI 를 추가할 수 있습니다. 이 GUI 는 `QgsRendererWidget` 클래스로부터 파생시켜야 합니다. 다음 코드는 여러분이 첫 번째 심볼을 설정할 수 있게 해주는 버튼을 생성하는 예시입니다:

```
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":
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
      self.r = RandomRenderer()
    else:
      self.r = renderer
    # setup UI
    self.btn1 = QgsColorButton()
    self.btn1.setColor(self.r.syms[0].color())
    self.vbox = QVBoxLayout()
    self.vbox.addWidget(self.btn1)
    self.setLayout(self.vbox)
    self.btn1.colorChanged.connect(self.setColor1)

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

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

이 작성자는 활성 레이어의 인스턴스 (`QgsVectorLayer`), 전체 수준 스타일 (`QgsStyle`), 그리고 현재 렌더링 작업자를 받습니다. 렌더링 작업자가 없거나 렌더링 작업자가 다른 유형인 경우, 새로운 사용자 정의 렌더링 작업자로 대체할 것입니다. 그렇지 않은 경우 (이미 필요한 유형인) 현재 렌더링 작업자를 사용할 것입니다. 렌더링 작업자의 현재 상태를 보이도록 위젯의 내용을 업데이트해야 합니다. 렌더링 작업자 대화창이 승인되면, 현재 렌더링 작업자를 가져오기 위해 위젯의 `renderer()` 메소드를 호출합니다—레이어에 가져온 렌더링 작업자를 할당할 것입니다.

마지막으로 렌더링 작업자의 메타데이터를 생성하고 레지스트리에 렌더링 작업자를 등록해야 합니다. 이렇게 하지 않으면 렌더링 작업자와 함께 레이어를 불러올 수 없고, 사용자가 렌더링 작업자 목록에서 렌더링 작업자를 선택할 수 없습니다. 이제 `RandomRenderer` 예시 코드를 완성해봅시다:

```
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)
```

심볼 레이어의 경우와 비슷하게, 추상 (abstract) 메타데이터 작성자도 렌더링 작업자 이름, 여러분이 볼 수 있는 이름, 그리고 선택적인 렌더링 작업자의 아이콘 이름을 기다립니다. `createRenderer()` 메소드가 DOM 트리로부터 렌더링 작업자의 상태를 복구하는 데 사용할 수 있는 `QDomElement` 클래스 인스턴스를 전달합니다. `createRendererWidget()` 메소드는 환경설정 위젯을 생성합니다. 렌더링 작업자가 GUI 를 가지고 있지 않은 경우, 이 메소드를 사용할 필요가 없고 또는 이 메소드가 `None` 을 반환할 수 있습니다.

렌더링 작업자에 아이콘을 연결하려면 `QgsRendererAbstractMetadata` 작성자에 아이콘을 세 번째 (선택적) 인자로 할당하면 됩니다—RandomRendererMetadata `__init__()` 함수에 있는 기저 클래스 작성자는 다음과 같이 보일 것입니다:

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

메타데이터 클래스의 `setIcon()` 메소드를 사용하면 이후 어느 때에라도 아이콘을 연결시킬 수 있습니다. 이 아이콘은 (앞에서 볼 수 있듯이) 파일에서 불러올 수도 있고, 또는 Qt 리소스 에서 불러올 수도 있습니다. (PyQt5 는 파이썬 용.qrc 컴파일러를 포함하고 있습니다.)

6.9 남은 이야기들

할 일:

- 심볼 생성/수정하기
- 스타일 작업하기 (`QgsStyle` 클래스)
- 색상표 작업하기 (`QgsColorRamp` 클래스)
- 심볼 레이어 및 렌더링 작업자 레지스트리 탐색하기

CHAPTER 7

도형 다루기

힌트

PyQGIS 콘솔을 사용하지 않는 경우 이 페이지에 있는 코드 조각들을 다음과 같이 가져와야 합니다:

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

공간 피처를 표현하는 포인트, 라인스트링, 그리고 폴리곤을 흔히 도형 (geometry) 이라고 부릅니다. QGIS 에서는 도형을 `QgsGeometry` 클래스를 사용해서 표현합니다.

도형 한 개가 실제로는 단순 (단일 부분, single-part) 도형의 집합인 경우가 종종 있습니다. 이런 도형을 다중 부분 (multi-part) 도형이라고 합니다. 다중 부분 도형이 한 가지 유형의 단순 도형으로만 이루어져 있을 경우 멀티포인트, 멀티라인스트링, 또는 멀티폴리곤이라고 부릅니다. 예를 들어 여러 개의 섬으로 이루어진 국가라면 멀티폴리곤으로 표현할 수 있습니다.

도형의 좌표는 어떤 좌표계라도 될 수 있습니다. 레이어에서 피처를 불러올 때, 해당 도형은 레이어의 좌표계를 따르는 좌표를 가지게 될 것입니다.

가능한 모든 도형 구조 및 관계에 대한 고급 상세 정보를 알고 싶다면 OGC 단순 피처 접근 표준 (OGC Simple Feature Access Standards) 에서 그 설명과 사양을 찾아볼 수 있습니다.

7.1 도형 작성하기

PyQGIS 는 도형을 생성하기 위한 옵션을 몇 개 제공합니다:

- 좌표로부터

```
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)
```

  `QgsPoint` 클래스 또는 `QgsPointXY` 클래스를 사용해서 좌표를 지정합니다. 이 두 클래스의 차이라고 하면 `QgsPoint` 클래스는 M 과 Z 차원을 지원한다는 점입니다.

  폴리라인 (라인스트링) 은 포인트 목록으로 표현됩니다.

  폴리곤은 선형 고리 (예: 닫힌 라인스트링들) 목록으로 표현됩니다. 첫 번째 고리 (ring) 가 외곽 고리 (경계) 이며, 있을 수도 있고 없을 수도 있는 그 다음 고리들은 폴리곤에 있는 구멍 (hole) 입니다. 몇몇 다른 프로그램과는 달리 QGIS 는 여러분을 위해 고리를 닫아줄 것이기 때문에 첫 번째 포인트를 마지막 포인트로 복제할 필요가 없다는 사실을 기억하세요.

  다중 부분 도형은 한 단계 심화됩니다. 멀티포인트는 포인트의 목록, 멀티라인스트링은 라인스트링의 목록, 멀티폴리곤은 폴리곤의 목록입니다.

- WKT(well-known text) 로부터

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

- WKB(well-known binary) 로부터

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

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

7.2 도형에 접근하기

먼저 도형 유형을 알아야 합니다. `wkbType()` 메소드를 사용하면 됩니다. 이 메소드는 `QgsWkbTypes.Type` 클래스의 열거 목록 (enumeration) 으로부터 값을 반환합니다.

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

아니면, `QgsWkbTypes.GeometryType` 메소드의 열거 목록으로부터 값을 반환하는 `type()` 메소드를 사용해도 됩니다.

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

사람이 읽을 수 있는 도형 유형을 얻으려면 `displayString()` 함수를 사용하십시오.

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

도형이 다중 부분인지 아닌지 여부를 알려주는 `isMultipart()` 도우미 함수도 있습니다.

모든 벡터 유형 별로 도형에서 정보를 추출하기 위한 접근자 (accessor) 함수들이 있습니다. 다음은 이런 접근자를 어떻게 사용하는지 보여주는 예시입니다:

```
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)>]]
```

> **참고**
>
> (x,y) 투플은 실제 투플 (tuple) 이 아니라 `QgsPoint` 클래스 객체로, `x()` 와 `y()` 메소드를 사용해서 그 값들에 접근할 수 있습니다.

다중 부분 도형의 경우 비슷한 접근자 함수들이 존재합니다: `asMultiPoint()`, `asMultiPolyline()`, 그리고 `asMultiPolygon()` 메소드들입니다.

도형 유형과는 상관없이 도형의 모든 부분들을 반복해서 작업할 수 있습니다. 예를 들면:

```
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)
```

`QgsGeometry.parts()` 메소드를 사용하면 도형의 각 부분을 수정할 수도 있습니다.

```
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 도형 술어 및 연산

QGIS 는 도형 술어 (predicate) 같은 고급 도형 연산 (`contains()`, `intersects()`, …) 과 집합 연산 (`combine()`, `difference()`, …) 에 GEOS 라이브러리를 사용합니다. GEOS 라이브러리는 (폴리곤의 경우) 면적 또는 (폴리곤 및 라인의 경우) 길이 같은 도형의 기하 속성도 계산할 수 있습니다.

지정한 레이어에 있는 피처들을 반복 작업해서 피처들의 도형을 바탕으로 도형 계산을 수행하는 예시를 볼까요. 다음 코드는 예제 QGIS 프로젝트 안에서 `countries` 레이어에 있는 각 국가의 면적과 둘레를 계산해서 출력할 것입니다.

다음 코드는 `layer` 가 폴리곤 피처 유형인 `QgsVectorLayer` 클래스 객체라고 가정합니다.

```
# 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
```

이제 도형들의 면적과 둘레를 계산해서 출력했습니다. 하지만 그 값들이 이상하다는 사실을 한 눈에 알아차렸겠죠. `QgsGeometry` 클래스에서 나온 `area()` 및 `length()` 메소드를 사용해서 면적과 둘레를 계산할 때 좌표계를

연산에 넣지 않았기 때문입니다. 좀 더 강력한 면적 및 거리 계산을 하고 싶다면, 타원체 기반 계산을 수행할 수 있는 `QgsDistanceArea` 클래스를 사용하면 됩니다:

다음 코드는 `layer` 가 폴리곤 피처 유형인 `QgsVectorLayer` 클래스 객체라고 가정합니다.

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

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

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

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

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

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

아니면, 포인트 2 개 사이의 거리를 알고 싶을 수도 있을 겁니다.

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

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

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

QGIS 에 포함되어 있는 알고리즘들의 수많은 예시를 찾아볼 수 있으며, 이 알고리즘들을 사용해서 벡터 데이터를 분석하고 변환할 수 있습니다. 다음 링크들은 그 가운데 몇몇 알고리즘들의 코드를 보여줍니다.

- `QgsDistanceArea` 클래스를 사용해서 거리 및 면적 계산: 거리 행렬 알고리즘
- 라인을 폴리곤으로 알고리즘

CHAPTER 8

투영체 지원

> 힌트
>
> PyQGIS 콘솔을 사용하지 않는 경우 이 페이지에 있는 코드 조각들을 다음과 같이 가져와야 합니다:

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

8.1 좌표계

`QgsCoordinateReferenceSystem` 클래스가 좌표계를 캡슐화합니다. 이 클래스의 인스턴스를 몇 가지 서로 다른 방법으로 생성할 수 있습니다:

- 좌표계 ID 로 좌표계 지정하기

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

```
True
```

QGIS 는 다음 서식을 가진 서로 다른 좌표계 식별자들을 지원합니다:

  - `EPSG:<code>` —EPSG 기관이 할당한 ID 입니다. `createFromOgcWms()` 메소드로 처리합니다.
  - `POSTGIS:<srid>`—PostGIS 데이터베이스에서 쓰이는 ID 입니다. `createFromSrid()` 메소드로 처리합니다.

- `INTERNAL:<srsid>` —QGIS 데이터베이스에서 쓰이는 ID 입니다. `createFromSrsId()` 메소드로 처리합니다.
- `PROJ:<proj>` —`createFromProj()` 메소드로 처리합니다.
- `WKT:<wkt>` —`createFromWkt()` 메소드로 처리합니다.

접두어를 지정하지 않는 경우, WKT(well-known text) 정의라고 가정합니다.

- WKT 로 좌표계 지정하기

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

```
True
```

- 무결하지 않은 좌표계를 생성한 다음 `create*` 함수들 가운데 하나를 사용해서 초기 설정합니다. 다음 예시에서는 투영체를 초기 설정하는 데 Proj 문자열을 사용합니다.

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

```
True
```

좌표계를 성공적으로 생성했는지 여부를 (예: 데이터베이스 검색) 확인하는 편이 좋습니다: `isValid()` 메소드가 `True` 를 반환해야만 합니다.

공간 참조 시스템 (spatial reference system) 을 초기 설정하기 위해 QGIS 가 `srs.db` 내부 데이터베이스에서 알맞은 값을 검색해야 한다는 사실을 기억하십시오. 즉 여러분이 독립 응용 프로그램을 생성한 경우 `QgsApplication.setPrefixPath()` 메소드로 경로를 정확하게 설정해야 합니다. 그렇지 않을 경우 이 데이터베이스를 찾지 못할 것입니다. 여러분이 QGIS 파이썬 콘솔에서 명령어를 실행하거나 또는 플러그인을 개발하는 중이라면 상관없습니다. 이미 모든 것이 설정되어 있기 때문입니다.

공간 참조 시스템 정보에 접근하기:

```
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())
```

산출물:

```
QGIS CRS ID: 3452
PostGIS SRID: 4326
Description: WGS 84
Projection Acronym: longlat
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Ellipsoid Acronym: EPSG:7030
Proj String: +proj=longlat +datum=WGS84 +no_defs
Is geographic: True
Map units: 6
```

8.2 좌표계 변환

QgsCoordinateTransform 클래스를 사용해서 서로 다른 공간 참조 시스템들을 변환시킬 수 있습니다. 이 클래스를 가장 쉽게 사용하는 방법은 소스 및 대상 좌표계를 생성한 다음 이 좌표계들과 현재 프로젝트를 가진 QgsCoordinateTransform 인스턴스를 작성하는 것입니다. 그 다음 transform() 함수를 반복 호출해서 변환을 수행합니다. 이 함수는 기본적으로 정방향 변환을 하지만, 역방향 변환도 할 수 있습니다.

```
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)
```

산출물:

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

CHAPTER 9

맵 캔버스 사용하기

힌트

PyQGIS 콘솔을 사용하지 않는 경우 이 페이지에 있는 코드 조각들을 다음과 같이 가져와야 합니다:

```
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,
)
```

맵 캔버스가 중첩하는 맵 레이어들로 구성된 맵을 보여주고 맵과 레이어들과 쌍방향 작업을 할 수 있게 해주기 때문에, 맵 캔버스 위젯은 아마도 QGIS 에서 가장 중요한 위젯일 겁다. 맵 캔버스는 항상 현재 캔버스 범위로 정의되는 맵의 일부분을 보여줍니다. 쌍방향 작업은 **맵 도구** 를 사용해서 이루어집니다: 이동 (pan), 확대/축소

(zoom), 레이어 식별, 측정, 벡터 편집, 그리고 기타 작업들을 위한 도구들이 있습니다. 다른 그래픽 프로그램들처럼, 도구 하나는 항상 활성화되어 있는 상태이며 여러분은 사용할 수 있는 다른 도구들로 전환할 수 있습니다.

맵 캔버스는 qgis.gui 모듈에 있는 `QgsMapCanvas` 클래스로 구현됩니다. 이 구현은 Qt 그래픽스 뷰 프레임워크를 기반으로 합니다. 이 프레임워크는 일반적으로 사용자 정의 그래픽 항목을 배치하고 쌍방향 작업을 할 수 있는 표면 (surface) 과 뷰 (view) 를 제공합니다. 여러분이 그래픽 신 (scene), 뷰, 항목이라는 개념을 충분히 이해할 만큼 Qt 에 익숙하다고 가정할 것입니다. 그렇지 않을 경우, 프레임워크의 개요 를 읽어보시기 바랍니다.

맵을 이동시키거나 확대/축소할 (또는 새로고침을 촉발하는 기타 다른 액션을 취할) 때마다, 현재 범위 안에 있는 맵을 다시 렌더링합니다. 레이어를 (`QgsMapRendererJob` 클래스를 사용해서) 이미지로 렌더링해서, 캔버스에 이 이미지를 보이는 것입니다. `QgsMapCanvas` 클래스는 렌더링된 맵의 새로고침도 제어합니다. 배경 역할을 하는 이 항목 외에, 더 많은 **맵 캔버스 항목들** 이 있을 수도 있습니다.

전형적인 맵 캔버스 항목이라고 하면 (측정, 벡터 편집 등등에 쓰이는) 고무줄 (rubber band) 또는 꼭짓점 마커 등등이 있습니다. 이런 캔버스 항목들은 보통 맵 도구들을 위한 가시적 피드백을 주기 위해 사용됩니다. 예를 들면 새 폴리곤을 생성할 때 맵 도구가 폴리곤의 현재 형태를 보여주는 고무줄 캔버스 항목을 생성합니다. 모든 맵 캔버스 항목은 기본 `QGraphicsItem` 객체에 더 많은 기능을 추가하는 `QgsMapCanvasItem` 클래스의 하위 클래스입니다.

요약하면 맵 캔버스 아키텍처는 다음 3 가지 개념으로 이루어집니다.

- 맵 캔버스—맵을 보여주는 데 쓰입니다.
- 맵 캔버스 항목—맵 캔버스에 보일 수 있는 추가적인 항목들입니다.
- 맵 도구—맵 캔버스와 쌍방향 작업을 하는 데 쓰입니다.

9.1 맵 캔버스 내장시키기

맵 캔버스는 다른 모든 Qt 위젯과 마찬가지이기 때문에, 이를 사용하는 방법도 생성하고 보이는 방법처럼 간단합니다.

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

이 코드는 맵 캔버스를 가진 독립형 (standalone) 창을 생성합니다. 기존 위젯이나 창에 맵 캔버스 창을 내장시킬 수도 있습니다. `.ui` 파일과 Qt 설계자를 사용할 때, 양식 상에 `QWidget` 을 배치하고 새 클래스로 승격시키십시오: `QgsMapCanvas` 를 클래스 이름으로 그리고 `qgis.gui` 를 헤더 파일로 설정하십시오. `pyuic5` 유틸리티가 이를 처리할 것입니다. 캔버스를 내장시킬 수 있는 매우 편리한 방법입니다. 다른 방법은 맵 캔버스와 기타 위젯들을 (메인 창 또는 대화창의 하위 창으로써) 작성하는 코드를 직접 입력한 다음 조판 (layout) 을 생성하는 것입니다.

맵 캔버스는 기본적으로 검은색 배경이고 위신호 제거 (anti-aliasing) 를 사용하지 않습니다. 배경을 하얀색으로 설정하고 매끄러운 렌더링을 위해 위신호 제거를 활성화하려면:

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

(여러분이 궁금할 경우를 위해 알려드리자면, `Qt` 는 `PyQt.QtCore` 모듈에서 나온 것이고 `Qt.white` 는 사전 정의된 `QColor` 인스턴스들 가운데 하나입니다.)

이제 맵 레이어 몇 개를 추가할 때입니다. 먼저 레이어 하나를 열어서 현재 프로젝트에 추가할 것입니다. 그 다음 캔버스 범위를 설정하고 캔버스에 보일 레이어 목록을 설정할 것입니다.

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

# 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])
```

이 명령어들을 실행하면, 사용자가 불러온 레이어가 캔버스에 보일 것입니다.

9.2 고무줄과 꼭짓점 마커

캔버스에 있는 맵 최상단에 추가 데이터를 보이려면 맵 캔버스 항목을 사용하십시오. (`writing-custom-map-canvas-items`_ 에서 설명할) 사용자 정의 캔버스 항목 클래스를 생성할 수 있지만, 사용자 편의를 위한 유용한 캔버스 항목 클래스 2 개가 있습니다: 폴리라인 또는 폴리곤을 그리기 위한 `QgsRubberBand` 클래스와 포인트를 그리기 위한 `QgsVertexMarker` 클래스입니다. 두 클래스 모두 맵 좌표를 사용해서 작업하기 때문에, 캔버스를 이동시키거나 확대/축소할 때 형태를 자동으로 이동/크기 조정합니다.

폴리라인을 보이려면:

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

폴리곤을 보이려면:

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

폴리곤의 포인트들이 평범한 목록이 아니라는 점을 기억하십시오. 사실 이 목록은 폴리곤의 선형 고리들을 담고 있는 고리 목록입니다. 첫 번째 고리는 외곽 경계이며, 그 다음 (있을 수도 있고 없을 수도 있는) 고리들은 폴리곤 내부의 구멍에 해당합니다.

고무줄을 사용자 정의할 수 있습니다. 즉 색상 및 선 너비를 변경할 수 있습니다.

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

캔버스 항목은 캔버스 신 (scene) 에 바인딩되어 있습니다. 캔버스 항목을 일시적으로 숨기려면 (그리고 다시 보이려면) `hide()` 와 `show()` 함수 조합을 사용하십시오. 어떤 항목을 완전히 제거하려면, 캔버스 신에서 제거해야 합니다:

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

(C++ 의 경우 항목을 그냥 삭제할 수 있지만, 파이썬의 경우 `del r` 명령어는 참조만 삭제할 뿐 실제 객체는 캔버스가 소유하고 있기 때문에 계속 남아 있을 것입니다.)

포인트도 고무줄을 사용해서 그릴 수 있지만, 이 작업에는 `QgsVertexMarker` 클래스가 더 적합합니다. (`QgsRubberBand` 클래스는 원하는 지점 주위에 직사각형을 그릴 뿐입니다.)

꼭짓점 마커는 다음과 같이 사용할 수 있습니다:

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

이 코드는 **[10,45]** 위치에 빨간색 십자표를 그릴 것입니다. 아이콘의 유형, 크기, 색상, 그리고 펜 너비를 사용자 정의할 수 있습니다:

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

꼭짓점 마커를 일시적으로 숨기거나 캔버스에서 제거하려면, 고무줄의 경우와 동일한 메소드들을 사용하십시오.

9.3 캔버스에서 맵 도구 사용하기

다음 예시는 맵 캔버스와 맵 이동 및 확대/축소를 위한 기본 맵 도구를 담고 있는 창일 작성합니다. 각 도구의 활성화를 위한 액션을 생성합니다: `QgsMapToolPan` 클래스는 이동을 수행하고, `QgsMapToolZoom` 클래스 인스턴스의 쌍으로 확대/축소를 수행합니다. 이 액션들을 체크할 수 있게 설정해서, 향후 액션의 체크/체크 해제 상태를 자동으로 처리할 수 있도록 도구에 할당합니다—어떤 맵 도구를 활성화시킬 때, 해당 도구의 액션을 선택 상태로 표시하고 이전 맵 도구의 액션을 선택 해제합니다. 맵 도구를 활성화시키는 메소드는 `setMapTool()` 입니다.

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

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

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

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

        self.setCentralWidget(self.canvas)

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

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

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

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

        # 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)
```

파이썬 콘솔 편집기에서 앞의 코드를 시도해볼 수 있습니다. 캔버스 창을 열려면, `MyWnd` 클래스를 초기 설정하는 다음 줄들을 추가하십시오. 새로 생성한 캔버스 위에 현재 선택된 레이어를 렌더링할 것입니다:

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

9.3.1 QgsMapToolIdentifyFeature 를 사용해서 피처 선택하기

사용자에게 콜백 (callback) 함수로 전송될 피처 하나를 선택하도록 요청하려면, `QgsMapToolIdentifyFeature` 맵 도구를 사용하면 됩니다.

```
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 맵 캔버스 컨텍스트 메뉴에 항목 추가하기

`contextMenuAboutToShow` 신호를 사용해서 맵 캔버스의 컨텍스트 메뉴에 추가할 수도 있는 항목들을 통해서도 맵 캔버스와의 쌍방향 작업을 수행할 수 있습니다.

다음은 맵 캔버스를 오른쪽 클릭할 때 나타나는 기본 항목에 *My menu* → *My Action* 액션을 추가하는 코드입니다.

```
# 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 사용자 정의 맵 도구 작성하기

여러분이 캔버스 상에서 수행하는 액션에 사용자 정의 습성을 구현하기 위한 사용자 정의 도구를 작성할 수 있습니다.

맵 도구는 클래스든 파생 클래스든 `QgsMapTool` 클래스로부터 상속받아야 하며, 이미 앞에서 본 바와 같이 캔버스에서 `setMapTool()` 메소드를 사용해서 활성 도구로써 선택되어야 합니다.

다음 예시는 캔버스 상에서 클릭 & 드래그로 직사각형 범위를 정의하도록 해주는 맵 도구의 코드입니다. 직사각형을 정의하면, 콘솔에 그 경계 좌표를 출력합니다. 앞에서 설명했던 고무줄 기능을 사용해서 정의되고 있는 직사각형을 보일 것입니다.

```
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()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
            )

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

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

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

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

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

  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 사용자 정의 맵 캔버스 항목 작성하기

다음은 원을 그리는 사용자 정의 캔버스 항목의 예시입니다:

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

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

  def center(self):
    return self.center

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  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)
```

CHAPTER 10

맵 렌더링 및 출력하기

힌트

이 페이지에 있는 코드 조각들을 다음과 같이 가져와야 합니다:

```python
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,
)
```

입력 데이터를 맵으로 렌더링해야 하는 경우 일반적으로 두 가지 접근법이 있습니다: `QgsMapRendererJob` 클래스를 사용해서 렌더링하는 빠른 방법 또는 맵을 `QgsLayout` 클래스로 구성해서 좀 더 미세 조정한 산출물을 생성하는 방법입니다.

10.1 단순 렌더링

렌더링 설정을 정의하는 `QgsMapSettings` 객체를 생성한 다음 해당 설정으로 `QgsMapRendererJob` 클래스를 작성해서 렌더링을 수행합니다. 후자는 산출 이미지를 생성하는 데 쓰입니다.

다음은 그 예시입니다:

```
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 서로 다른 좌표계를 가진 레이어들을 렌더링하기

레이어가 하나 이상 있는데 그 레이어들이 서로 다른 좌표계를 가지고 있는 경우, 앞의 단순 예시는 아마도 작동하지 않을 것입니다. 범위 계산으로부터 올바른 값들을 얻으려면 대상 좌표계를 명확하게 설정해야 합니다:

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

10.3 인쇄 조판을 사용해서 산출하기

앞에서 본 단순 렌더링보다 좀 더 세련된 산출물을 생성하고 싶은 경우, 인쇄 조판 (print layout) 이 매우 유용한 도구입니다. 맵 뷰, 라벨, 범례, 표 그리고 보통 종이 지도에서 볼 수 있는 기타 요소들로 이루어진 복잡한 맵 조판을 생성할 수 있습니다. 나중에 이 조판을 PDF, SVG, 래스터 이미지로 내보내거나 프린터에서 직접 인쇄할 수 있습니다.

조판은 클래스 여러 개들로 이루어집니다. 이 클래스들은 모두 핵심 라이브러리에 속해 있습니다. QGIS 응용 프로그램은 이런 요소들을 배치하기 위한 편리한 GUI 를 가지고 있지만, GUI 라이브러리에서 이 GUI 를 사용할 수는 없습니다. 여러분이 Qt 그래픽스 뷰 프레임워크 에 익숙하지 않은 경우, 조판이 이 프레임워크를 기반으로 하기 때문에, 지금 해당 문서를 읽어볼 것을 권장합니다.

조판의 중심 클래스는 Qt QGraphicsScene 클래스에서 파생된 `QgsLayout` 클래스입니다. 이 클래스의 인스턴스를 생성해봅시다:

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

이 코드는 몇몇 기본 설정으로, 그 중에서도 조판에 비어 있는 A4 페이지를 추가해서 조판을 초기 설정합니다. `initializeDefaults()` 메소드를 호출하지 않고서도 조판을 생성할 수 있지만, 여러분이 조판에 페이지를 직접 추가해야 할 것입니다.

앞의 코드는 GUI 에 가시화되지 않는"임시"조판을 생성합니다. 프로젝트 자체를 수정하지 않고, 또는 사용자에게 이런 변경 사항을 노출시키지 않고서도 몇몇 항목들을 빠르게 추가하고 내보낼 수 있는 편리한 방법입니다. 조판을 프로젝트와 함께 저장하거나 복원하고 조판 관리자에서 사용할 수 있기를 바라는 경우 다음 코드를 추가하십시오:

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

이제 조판에 여러 가지 (맵, 라벨, …) 요소들을 추가할 수 있습니다. 이 객체들은 모두 기저 `QgsLayoutItem` 클래스에서 상속받은 클래스들로 표현됩니다.

다음은 조판에 추가할 수 있는 주요 조판 항목들 가운데 일부를 설명합니다.

- map —사용자 정의 크기를 가진 맵을 생성하고 현재 맵 캔버스를 렌더링하는 코드입니다:

```
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 —라벨을 보이게 할 수 있게 해주는 코드입니다. 글꼴, 색상, 정렬 및 여백을 수정할 수 있습니다:

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

- legend

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

- 축척막대 (scale bar)

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

- 노드 기반 형태

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

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

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

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

조판에 항목을 추가하고 나면, 항목을 이동시키고 크기를 조정할 수 있습니다:

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

기본적으로 각 항목 주위에 프레임을 그립니다. 이 프레임을 다음과 같이 제거할 수 있습니다:

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

조판 항목을 직접 생성하는 방법 이외에, QGIS 는 본질적으로 `.qpt` 파일에 모든 조판 항목들을 (XML 문법을 사용해서) 저장한 조판 (composition) 인 조판 템플릿을 지원합니다.

이 조판이 준비되고 나면 (조판 항목들을 생성해서 조판에 추가하고 나면) 래스터 그리고/또는 벡터 산출물 생성을 시작할 수 있습니다.

10.3.1 조판 무결성 검증하기

조판은 상호 연결된 항목들의 집합으로 이루어져 있는데, 수정 작업 동안 이 연결이 깨지는 경우 (제거한 맵에 연결된 범례, 누락된 소스 파일을 가진 이미지 항목 등등) 가 있을 수도 있고, 또는 여러분이 조판 항목에 사용자 정의 제약조건을 적용시키고 싶을 수도 있습니다. 이런 경우 `QgsAbstractValidityCheck` 클래스가 도움이 될 것입니다.

기본 검증 코드는 다음처럼 보일 겁니다:

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

다음은 조판 맵 항목이 웹 메르카토르 투영법으로 설정되어 있을 때마다 경고 메시지를 띄우는 검증 코드의 예시입니다:

```
@check.register(type=QgsAbstractValidityCheck.TypeLayoutCheck)
def layout_map_crs_choice_check(context, feedback):
  layout = context.layout
  results = []
  for i in layout.items():
    if isinstance(i, QgsLayoutItemMap) and i.crs().authid() == 'EPSG:3857':
      res = QgsValidityCheckResult()
      res.type = QgsValidityCheckResult.Warning
      res.title = 'Map projection is misleading'
      res.detailedDescription = 'The projection for the map item {} is set to <i>Web␣
↪Mercator (EPSG:3857)</i> which misrepresents areas and shapes. Consider using an␣
↪appropriate local projection instead.'.format(i.displayName())
      results.append(res)

  return results
```

그리고 다음은 더 복잡한 예시입니다. 이 코드는 조판 맵 항목이 해당 맵 항목의 범위 바깥에서만 무결한 좌표계로 설정되어 있는 경우 경고 메시지를 띄웁니다:

```
@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 조판 내보내기

조판을 내보내려면, 반드시 `QgsLayoutExporter` 클래스를 사용해야만 합니다.

```
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())
```

여러분이 조판을 SVG 또는 이미지 파일로 아니면 PDF 파일로 내보내고 싶은 경우 각각 `exportToSvg()` 아니면 `exportToImage()` 메소드를 사용하십시오.

10.3.3 조판 지도책 내보내기

지도책 (atlas) 옵션이 환경설정되어 있고 활성화된 조판에서 모든 페이지를 내보내고자 하는 경우, 내보내기 작업자 (`QgsLayoutExporter` 클래스) 에서 `atlas()` 메소드를 약간 조정해서 사용해야 합니다. 다음 예시는 페이지들을 PNG 이미지로 내보내는 코드입니다:

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

산출물이 기본 경로 폴더에 지도책에 대해 환경설정된 산출 파일 이름 표현식을 사용해서 저장될 것이라는 사실을 기억하십시오.

CHAPTER 11

값을 필터링하고 계산하는 표현식

힌트

PyQGIS 콘솔을 사용하지 않는 경우 이 페이지에 있는 코드 조각들을 다음과 같이 가져와야 합니다:

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

QGIS 는 SQL 계열 표현식의 파싱을 일부분 지원합니다. SQL 문법 가운데 작은 부분만을 지원합니다. (`True` 또는 `False` 를 반환하는) 불 술어 (boolean predicate) 또는 (스칼라 값을 반환하는) 함수 가운데 하나로 표현식을 평가할 수 있습니다. 사용할 수 있는 함수의 전체 목록을 보고 싶다면 사용자 지침서에 있는 vector_expressions 을 참조하세요.

다음 기본 유형 3 개를 지원합니다:

- 숫자—정수와 십진수, 예: `123`, `3.14`
- 문자열—`'hello world'` 처럼 작은 따옴표로 둘러싸야 합니다.
- 열 (column) 참조—표현식을 평가할 때, 참조를 필드의 실제 값으로 대체합니다. 이름은 이스케이프시키지 않습니다.

다음과 같은 연산자들을 사용할 수 있습니다:

- 산술 연산자: +, -, *, /, ^
- 괄호: `(1 + 1) * 3` 처럼 연산의 우선 순위를 강제합니다.
- 단항 플러스 및 마이너스: `-12`, `+5`
- 수학 함수: `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
- 변환 함수: `to_int`, `to_real`, `to_string`, `to_date`
- 도형 함수: `$area`, `$length`
- 도형 처리 함수: `$x`, `$y`, `$geometry`, `num_geometries`, `centroid`

다음과 같은 술어들을 지원합니다:

- 비교: =, !=, >, >=, <, <=
- 패턴 매칭: `LIKE` (`%` 와 _ 사용), ~ (정규 표현식)
- 논리 술어: `AND`, `OR`, `NOT`
- NULL 값 검증: `IS NULL`, `IS NOT NULL`

술어의 예:

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

스칼라 표현식의 예:

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

11.1 표현식 파싱하기

다음은 지정한 표현식을 정확하게 파싱할 수 있는지 여부를 검증하는 방법을 보여주는 예시입니다:

```
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 표현식 평가하기

서로 다른 맥락에서, 예를 들면 피처를 필터링하기 위해 또는 새 필드 값을 계산하기 위해 표현식을 사용할 수 있습니다. 어떤 경우라도, 표현식을 평가해야 합니다. 즉 단순 산술 표현식에서 집계 표현식까지 아우를 수 있는 범위에서, 지정된 계산 단계를 따라 그 값을 계산한다는 뜻입니다.

11.2.1 기본 표현식

다음 기본 표현식은 단순 산술 연산을 평가합니다:

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

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

비교에도 1(`True`) 또는 0(`False`) 으로 평가하는 표현식을 사용할 수 있습니다:

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

11.2.2 피처를 사용하는 표현식

피처에 대한 표현식을 평가하려면, 표현식이 피처의 필드 값에 접근할 수 있도록 `QgsExpressionContext` 클래스 객체를 생성해서 평가 함수로 전달해야 합니다.

다음은"Column"이라는 필드를 가진 피처를 생성하는 방법과 표현식 맥락에서 해당 피처를 추가하는 방법을 보여주는 예시입니다:

```
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
```

다음은 새 필드 값을 계산하기 위해 벡터 레이어 맥락에서 표현식을 사용하는 방법을 보여주는 좀 더 복잡한 예시입니다:

```
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),
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                    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
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 표현식으로 레이어 필터링하기

레이어를 필터링해서 술어와 일치하는 모든 피처를 반환하는 데 다음 예시를 사용할 수 있습니다:

```
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 표현식 오류 처리하기

표현식을 파싱하거나 평가하는 동안 표현식 관련 오류가 발생할 수 있습니다:

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

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

CHAPTER 12

읽기 및 저장하기 설정

힌트

PyQGIS 콘솔을 사용하지 않는 경우 이 페이지에 있는 코드 조각들을 다음과 같이 가져와야 합니다:

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

플러그인에 몇몇 변수를 저장하는 것이 유용한 경우가 많습니다. 여러분이 다음에 해당 플러그인을 실행할 때 변수들을 다시 입력하거나 선택하지 않아도 되기 때문입니다.

Qt 및 QGIS API 덕분에 이런 변수들을 저장하고 가져올 수 있습니다. 각 변수에 대해, 해당 변수에 접근하는 데 사용할 키를 선택해야 합니다—예를 들면 여러분이 가장 선호하는 색상에 대해"favourite_color"또는 비슷한 의미를 가진 다른 어떤 문자열도 사용할 수 있습니다. 키를 명명할 때 어떤 기준을 따를 것을 권장합니다.

설정을 다음 몇 가지 서로 다른 유형으로 구분할 수 있습니다:

- **전체 수준 설정 (global settings)** –이 설정은 특정 머신의 사용자에 종속됩니다. QGIS 자체도 수많은, 예를 들면 메인 창의 크기 또는 기본 스냅 작업 허용 오차와 같은 전체 수준 설정을 저장합니다. 예를 들어 `setValue()` 및 `value()` 메소드를 통해 `QgsSettings` 클래스를 사용해서 설정을 처리합니다.

  다음은 이런 메소드들을 어떻게 사용하는지 보여주는 예시입니다.

```
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()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

`value()` 메소드의 두 번째 파라미터는 선택적으로, 전달된 설정 이름에 대해 이전에 설정한 값이 없을 경우 반환하는 기본값을 지정합니다.

`qgis_global_settings.ini` 파일을 통해 전체 수준 설정들의 기본값을 사전 환경설정할 수 있는 메소드에 대해 알고 싶다면 deploying_organization 를 참조하십시오.

- **프로젝트 수준 설정**—프로젝트에 따라 달라지기 때문에 프로젝트 파일과 연동됩니다. 프로젝트 수준 설정의 예시로는 맵 캔버스 배경 색상 또는 대상 좌표계 등을 들 수 있습니다. 어떤 프로젝트에서는 하얀 배경과 WGS84 가 어울리는 반면 다른 프로젝트에서는 노란 배경과 UTM 투영 좌표계가 어울리는 경우가 있는 법이죠.

  다음은 프로젝트 수준 설정 사용의 예시입니다.

```
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)
```

  여러분도 알 수 있듯이, 많은 (정수, 문자열, 목록) 데이터 유형에 대해 `writeEntry()` 메소드를 사용하지만, 설정값을 다시 읽어오기 위한 메소드가 몇 가지 존재하며 각 데이터 유형에 대응하는 메소드를 선택해야 합니다.

- **맵 레이어 수준 설정**—프로젝트의 맵 레이어 하나의 특정 인스턴스와 관련된 설정입니다. 이 설정은 레이어의 기저 데이터 소스와 연결되지 않기 때문에, 셰이프파일 (shapefile) 1 개의 맵 레이어 인스턴스를 2 개 생성하는 경우 이 두 인스턴스는 설정을 공유하지 않을 것입니다. 맵 레이어 수준 설정은 프로젝트 파일 내부에 저장되기 때문에 여러분이 프로젝트를 다시 열면 레이어 관련 설정도 그대로 다시 불러올 것입니다. `customProperty()` 메소드를 사용해서 지정한 설정의 값을 검색하고, `setCustomProperty()` 메소드를 사용해서 설정할 수 있습니다.

```
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")
```

CHAPTER 13

사용자에게 정보 전달하기

힌트

PyQGIS 콘솔을 사용하지 않는 경우 이 페이지에 있는 코드 조각들을 다음과 같이 가져와야 합니다:

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

from qgis.gui import (
    QgsMessageBar,
)

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

이 장에서는 사용자 인터페이스에서 일관성을 유지하기 위해 사용자에게 정보를 전달하는 데 사용해야 하는 몇몇 메소드 및 요소에 대해 설명합니다.

13.1 메시지 보이기: QgsMessageBar 클래스

메시지 상자를 사용하는 것은 사용자 경험이라는 관점에서 그리 좋은 생각이 아닙니다. 짧은 정보 또는 경고/오류 메시지를 표출하는 데에는 QGIS 메시지 바를 사용하는 편이 더 낫습니다.

QGIS 인터페이스 객체에 참조 (reference) 를 사용하면, 메시지 바에 메시지를 다음 코드로 표시할 수 있습니다:

```
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
```

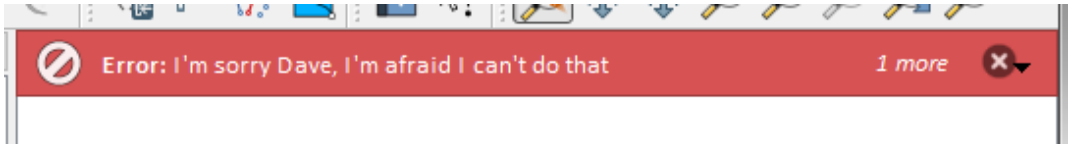


그림 13.1: QGIS 메시지 바

제한 시간 동안에만 메시지를 보이려면 지속 시간 (`duration`) 을 설정하면 됩니다.

```
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
```

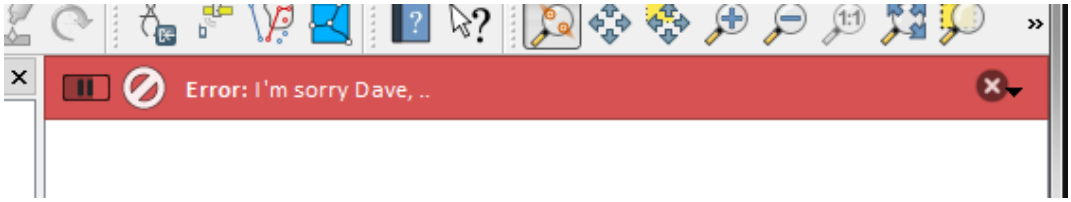


그림 13.2: 타이머를 적용한 QGIS 메시지 바

앞의 예시들은 오류 메시지를 보이지만, `level` 파라미터를 사용하면 `Qgis.MessageLevel` 클래스의 열거 목록 (enumeration) 을 통해 경고 또는 정보 메시지를 생성할 수 있습니다. 모두 4 개의 서로 다른 수준을 사용할 수 있습니다:

0. 정보 (Info)
1. 경고 (Warning)
2. 중요 (Critical)
3. 성공 (Success)

추가적인 정보를 표시하기 위한 버튼 등의 위젯을 메시지 바에 추가할 수도 있습니다:

```
def showError():
    pass

widget = iface.messageBar().createMessage("Missing Layers", "Show Me")
button = QPushButton(widget)
```

(다음 페이지에 계속)

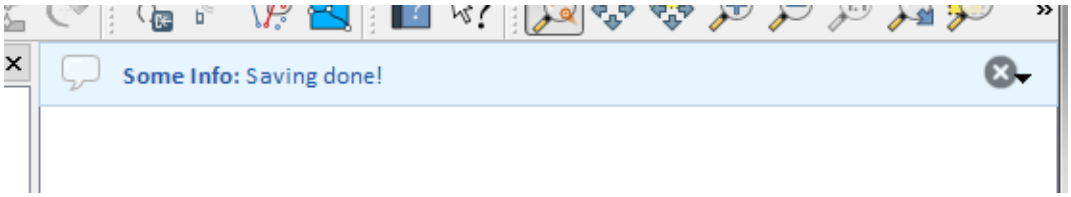


그림 13.3: QGIS 메시지 바 (정보)

(이전 페이지에서 계속)

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

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

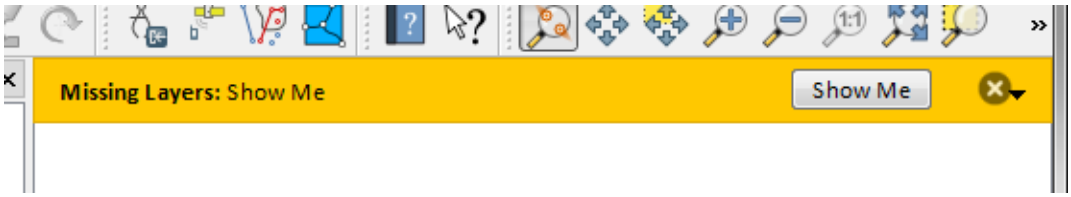


그림 13.4: 버튼이 들어간 QGIS 메시지 바

별도의 메시지 상자가 필요 없거나 QGIS 메인 창에 메시지를 표시할 이유가 없을 경우, 사용자의 대화창에 메시지 바를 표시할 수도 있습니다.

```
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()
```

그림 13.5: 사용자 정의 대화창 내부의 QGIS 메시지 바

13.2 진행률 보이기

앞에서 배웠듯이 QGIS 메시지 바에 위젯을 추가할 수 있으므로, 진행률 (progress) 막대도 추가할 수 있습니다. 다음은 여러분이 콘솔에서 실행해볼 수 있는 예시 코드입니다.

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

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

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

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

또한 다음 예시에서와 같이 진행률을 보고하는 내장 상태 바도 사용할 수 있습니다:

```
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 로그 작성하기

여러분의 코드를 실행하는 것에 대한 모든 정보를 로그로 작성하고 저장할 수 있도록, QGIS 는 세 가지의 서로 다른 로그 작업 유형을 지원합니다. 각 유형은 전용 산출 위치를 가지고 있습니다. 여러분의 목적에 따라 올바른 로그 작업 방식을 사용해주십시오:

- `QgsMessageLog` 는 사용자에게 문제점을 전달하기 위한 메시지 용 클래스입니다. QgsMessageLog 클래스의 산출물은 로그 메시지 패널에 나타납니다.
- 파이썬 내장 **logging** 모듈은 QGIS 파이썬 API (PyQGIS) 수준에서 디버그 작업을 하기 위한 모듈입니다. 예를 들면 피처 ID 또는 도형 같은 파이썬 코드를 디버그해야 하는 파이썬 스크립트 개발자들에게 권장합니다.
- `QgsLogger` 는 (예를 들어 어떤 깨진 코드가 뭔가를 촉발한다고 의심하는 경우) *QGIS* 내부 디버그 작업 또는 개발자 용 클래스입니다. 이 메시지는 QGIS 개발자 버전에서만 보일 것입니다.

다음 절들에서 이 서로 다른 로그 작업 유형들의 예시를 볼 수 있습니다.

> **경고**
>
> 파이썬 `print` 선언문을 사용하면 멀티스레드일 수도 있는 모든 코드를 안전하지 못 하게 만들고 **알고리즘을 극단적으로 느려지게 합니다.** 이때 모든 코드란 (다른 것들 가운데에서도) **표현식 함수**, **렌더링 작업자**, **심볼 레이어**, 그리고 **공간 처리 알고리즘** 을 포함합니다. 이런 경우 항상 파이썬 **로그 작업** 모듈 또는 스레드 안전 클래스들 (`QgsLogger` 또는 `QgsMessageLog`) 을 대신 시용해야 합니다.

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!
```

> **참고**
>
> log_message_panel 에서 `QgsMessageLog` 클래스의 산출물을 볼 수 있습니다.

13.3.2 파이썬 내장 로그 작업 모듈

```
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 메소드가 로그 작업의 기본 설정을 환경설정합니다. 앞의 코드에서 파일 이름, 로그 작업 수준 및 서식을 정의했습니다. 파일 이름은 로그 파일을 작성할 위치를 알려주며, 로그 작업 수준은 산출할 수준을 정의하고, 서식은 각 메시지를 산출할 서식을 정의합니다.

```
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
```

여러분이 여러분의 스크립트를 실행할 때마다 로그 파일을 지우길 바라는 경우, 다음과 같은 코드를 작성하면 됩니다:

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

다음 링크에서 파이썬 로그 작업 기능을 사용하는 방법에 대한 심화 자료들을 찾아볼 수 있습니다:

- 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

경고

파일 이름을 설정해서 파일에 로그를 작성하지 않는다면 로그 작업이 산출 속도를 크게 떨어뜨리는 멀티스레드가 될 수도 있다는 사실을 기억하십시오.

CHAPTER 14

인증 인프라스트럭처

힌트

PyQGIS 콘솔을 사용하지 않는 경우 이 페이지에 있는 코드 조각들을 다음과 같이 가져와야 합니다:

```
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 소개

사용자 지침서의 authentication_overview 문단에서 인증 인프라스트럭처의 사용자 참조 사항들을 읽어볼 수 있습니다.

이 장에서는 개발자 입장에서 이 인증 시스템을 가장 잘 활용할 수 있는 방법을 설명할 것입니다.

어떤 특정 리소스에 접근하려면 인증 정보가 필요할 때마다, 예를 들어 PostgreSQL 데이터베이스에 어떤 레이어를 연결할 때마다, 데이터 제공자는 QGIS 데스크탑에서 인증 시스템을 광범위하게 사용합니다.

플러그인 개발자가 자신의 코드에 인증 인프라스트럭처를 손쉽게 통합할 수 있도록 해주는 위젯 몇 개가 QGIS GUI 라이브러리에 존재합니다:

- `QgsAuthConfigEditor`
- `QgsAuthConfigSelect`
- `QgsAuthSettingsWidget`

인증 인프라스트럭처 테스트 코드 에서 괜찮은 코드 참조 사례를 볼 수 있습니다.

> **경고**
>
> 인증 인프라스트럭처 설계 시 도입된 보안 제약조건들 때문에, 파이썬에서는 내부 메소드들 가운데 일부 하위 집합만을 사용할 수 있습니다.

14.2 용어 해설

이 장에서 가장 자주 취급하는 객체들을 몇 개 정의하겠습니다.

마스터 비밀번호
: QGIS 인증 데이터베이스에 저장되어 있는 인증 정보 (credential) 에 접근해서 복호화할 수 있게 해주는 비밀번호입니다.

인증 데이터베이스
: *Master Password* 가 암호화되어 있는 SQLite 데이터베이스 `qgis-auth.db` 입니다. 이 데이터베이스에 사용자 이름/비밀번호, 개인 인증서와 키, 인증서 기관 (Certificate Authorities) 같은*Authentication Configuration* 이 저장되어 있습니다.

인증 DB
: *Authentication Database*

인증 환경설정 (Authentication Configuration)
: *Authentication Method* 에 따라 달라지는 인증 데이터 집합입니다. 예를 들면 기본 인증 방법은 사용자 이름/비밀번호 쌍을 저장합니다.

인증 환경설정 (Authentication Config)
: *Authentication Configuration*

인증 방법
: 인증을 받기 위해 쓰이는 특정 방법입니다. 각 방법은 인증 수준을 얻는 데 사용되는 고유 프로토콜을 가지고 있습니다. 각 방법은 QGIS 인증 인프라스트럭처 초기 설정 시 동적으로 불러오는 공유 라이브러리로 구현됩니다.

14.3 QgsAuthManager: 입구점

`QgsAuthManager` 클래스 단독 개체 (singleton) 는 예를 들어 활성 사용자 프로파일 폴더 아래 있는 `qgis-auth.db` 파일 같은 QGIS 암호화*Authentication DB* 에 저장되어 있는 인증 정보를 사용하기 위한 입구점 (entry point) 입니다.

이 클래스는 저장된 암호화 정보에 접근하기 위해 마스터 비밀번호를 설정하라고 요청하거나 또는 마스터 비밀번호를 투명하게 사용하는 방식으로 사용자의 쌍방향 작업을 처리합니다.

14.3.1 관리자 초기 설정 및 마스터 비밀번호 설정하기

다음 코드 조각은 인증 설정에 접근하기 위한 마스터 비밀번호를 설정하는 예시입니다. 이 코드 조각을 이해하려면 코드 주석을 잘 읽어보십시오.

```
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 새 인증 환경설정 항목으로 인증 DB 채우기

저장된 모든 인증 정보는 다음과 같은 유일한 문자열을 사용해서 접근하는 `QgsAuthMethodConfig` 클래스의*Authentication Configuration* 인스턴스입니다:

```
authcfg = 'fm1s770'
```

QGIS API 또는 GUI 를 사용해서 환경설정 항목을 생성할 때 이런 문자열을 자동으로 생성하지만, 어떤 조직 내부에서 사용자 여러 명 사이에서 인증 환경설정을 (서로 다른 인증 정보를 사용해서) 공유해야만 하는 경우 이미 알고 있는 값으로 직접 설정하는 편이 유용할 수도 있습니다.

`QgsAuthMethodConfig` 클래스는 모든*Authentication Method* 의 기저 클래스입니다. 모든 인증 방법은 인증 정보를 저장할 환경설정 해시 맵 (configuration hash map) 을 설정합니다. 다음은 `alice` 라는 가상의 사용자에 대한 PKI 경로 인증 정보를 저장할 수 있는 유용한 코드 조각입니다:

```
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
```

사용할 수 있는 인증 방법들

인증 관리자 초기 설정 시*Authentication Method* 라이브러리를 동적으로 불러옵니다. 다음 인증 방법을 사용할 수 있습니다:

1. `Basic`: 사용자 이름과 비밀번호 인증
2. `EsriToken`: ESRI 토큰 기반 인증
3. `Identity-Cert`: 신원 인증서 인증
4. `OAuth2`: OAuth2(Open Authorization 2.0) 인증
5. `PKI-Paths`: PKI 경로 인증
6. `PKI-PKCS#12`: PKI PKCS#12 인증

기관 채우기

```
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 을 사용해서 PKI 번들 관리하기

`QgsPkiBundle` 클래스는 SSLCert, SSLKey 및 CA 체인에 대해 구성된 PKI 번들을 패키징하는 편리한 클래스입니다. 다음은 비밀번호를 보호하기 위한 코드 조각입니다:

```
# 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()
```

이 번들에서 인증서/키/인증 기관을 추출하려면 `QgsPkiBundle` 클래스 문서를 참조하세요.

14.3.3 인증 DB 에서 항목 제거하기

Authentication Database 의 `authcfg` 식별자를 다음 코드 조각처럼 사용하면 인증 데이터베이스에서 항목 하나를 제거할 수 있습니다:

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

14.3.4 QgsAuthManager 에 인증 환경설정 확장을 맡기기

Authentication DB 에 저장되어 있는*Authentication Config* 을 사용하는 최선의 방법은 `authcfg` 식별자를 사용해서 인증 환경설정을 참조하는 것입니다. 확장이란 이 식별자 하나를 완전한 인증 정보 집합으로 변환하는 것을 말합니다. 저장되어 있는*Authentication Config* 들을 사용하는 최선의 방법은 인증 관리자가 자동으로 관리하도록 내버려두는 것입니다. WMS 또는 WFS 처럼 인증이 활성화된 서비스 또는 데이터베이스에 연결하는 것이 저장된 환경설정을 사용하는 가장 흔한 방법입니다.

> **참고**
>
> 모든 QGIS 데이터 제공자가 인증 인프라스트럭처와 통합되어 있지는 않다는 사실을 기억하십시오. `QgsAuthMethod` 기저 클래스에서 파생된 각 인증 방법은 서로 다른 제공자들을 지원합니다. 예를 들어 `certIdentity()` 메소드는 다음 제공자들을 지원합니다:
>
> ```
> authM = QgsApplication.authManager()
> print(authM.authMethod("Identity-Cert").supportedDataProviders())
> ```
>
> 이 예시의 산출물:
>
> ```
> ['ows', 'wfs', 'wcs', 'wms', 'postgres']
> ```

예를 들면, `authcfg = 'fm1s770'` 문자열로 식별되는 저장된 인증 정보를 사용해서 WMS 서비스에 접근하려는 경우 다음 코드 조각에서처럼 데이터 소스 URL 에 `authcfg` 를 사용하기만 하면 됩니다:

```
authCfg = 'fm1s770'
quri = QgsDataSourceUri()
quri.setParam("layers", 'usa:states')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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')
```

앞의 예시에서, `wms` 제공자가 HTTP 연결을 설정하기 직전에 `authcfg` URI 파라미터를 인증 정보를 사용해서 확장시킬 것입니다.

> **경고**
>
> 개발자는 `authcfg` 확장을 `QgsAuthManager` 클래스에 맡겨야 할 것입니다. 이렇게 하면 너무 일찍 확장되지 않을 것을 믿을 수 있기 때문입니다.

`QgsDataSourceURI` 클래스를 사용해서 작성된 URI 문자열은 보통 다음 방식으로 데이터 소스를 설정하는 데 쓰입니다:

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

> **참고**
>
> `False` 파라미터가 URI 에 존재하는 `authcfg` ID 의 URI 완전 확장을 방지하는 데 중요한 역할을 합니다.

기타 데이터 제공자들의 PKI 예시

QGIS 테스트 업스트림에 있는 test_authmanager_pki_ows 또는 test_authmanager_pki_postgres 에서 기타 예시를 직접 읽어볼 수 있습니다.

14.4 플러그인이 인증 인프라스트럭처를 사용하도록 조정하기

제 3 자 플러그인 가운데 다수가 HTTP 연결을 관리하기 위해 `QgsNetworkAccessManager` 클래스 및 관련 통합 인증 인프라스트럭처와 통합하는 대신 httplib2 또는 다른 파이썬 네트워크 작업 라이브러리를 사용하고 있습니다.

이런 통합을 용이하게 하기 위해 `NetworkAccessManager` 라는 파이썬 도우미 함수가 생성되었습니다. 여기에서 그 코드를 찾아볼 수 있습니다.

이 도우미 클래스를 다음 코드 조각에서처럼 사용할 수 있습니다:

```
http = NetworkAccessManager(authid="my_authCfg", exception_class=My_
↪FailedRequestError)
try:
  response, content = http.request( "my_rest_url" )
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

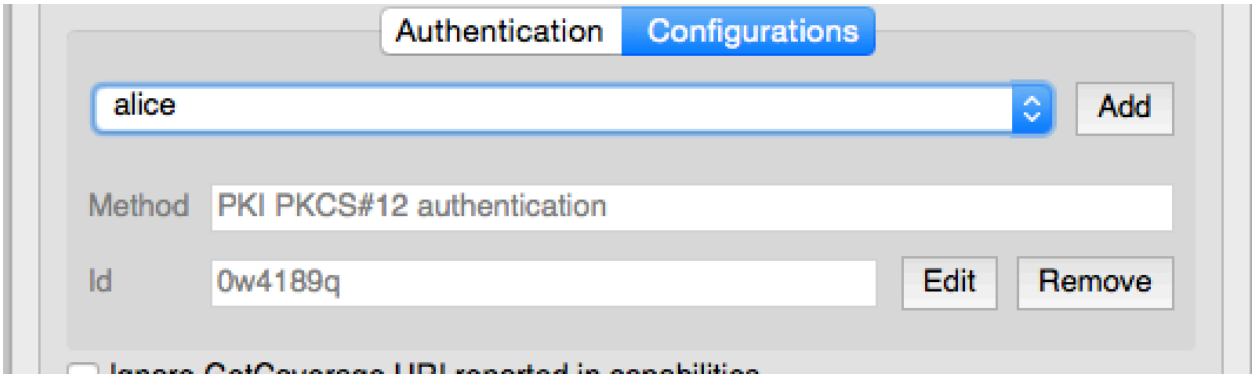
```
except My_FailedRequestError, e:
  # Handle exception
  pass
```

14.5 인증 GUI

이 절에서는 사용자 정의 인터페이스에 인증 인프라스트럭처를 통합하는 데 사용할 수 있는 유용한 GUI 들을 소개합니다.

14.5.1 인증 정보를 선택할 수 있는 GUI

Authentication DB 에 저장된 집합에서*Authentication Configuration* 하나를 선택해야 하는 경우 `QgsAuthConfigSelect` GUI 클래스에서 선택할 수 있습니다.



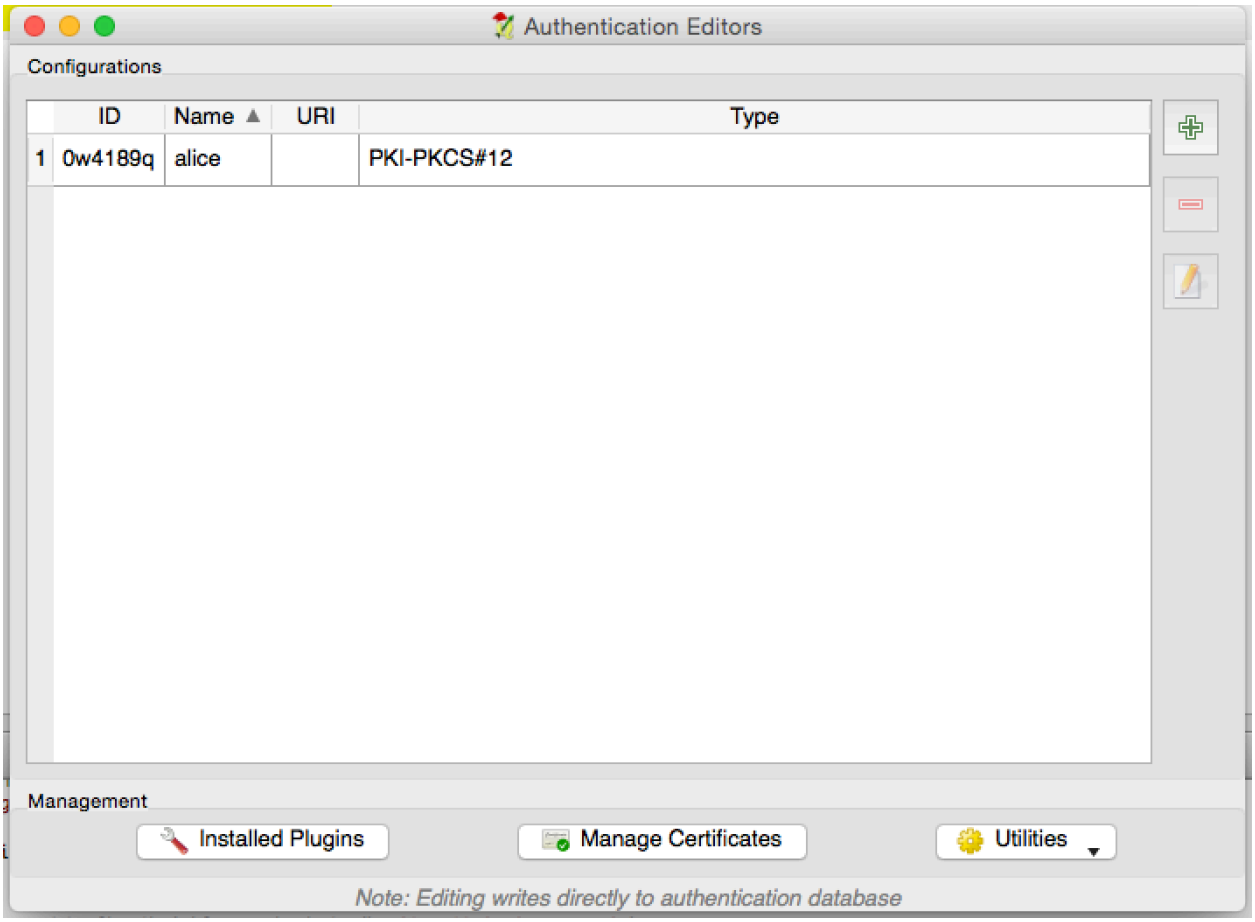
다음 코드 조각에서와 같이 사용하면 됩니다:

```
# 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" )
```

앞의 예시는 QGIS 소스 코드 에서 발췌한 것입니다. GUI 작성자의 두 번째 파라미터는 데이터 제공자 유형을 알려줍니다. 이 파라미터는 지정한 제공자와 호환되는*Authentication Method* 들을 제한하기 위해 쓰입니다.

14.5.2 인증 편집기 GUI

`QgsAuthEditorWidgets` 클래스가 인증 정보, 기관을 관리하고 인증 유틸리티들에 접근하는 데 사용되는 완전한 GUI 를 관리합니다.



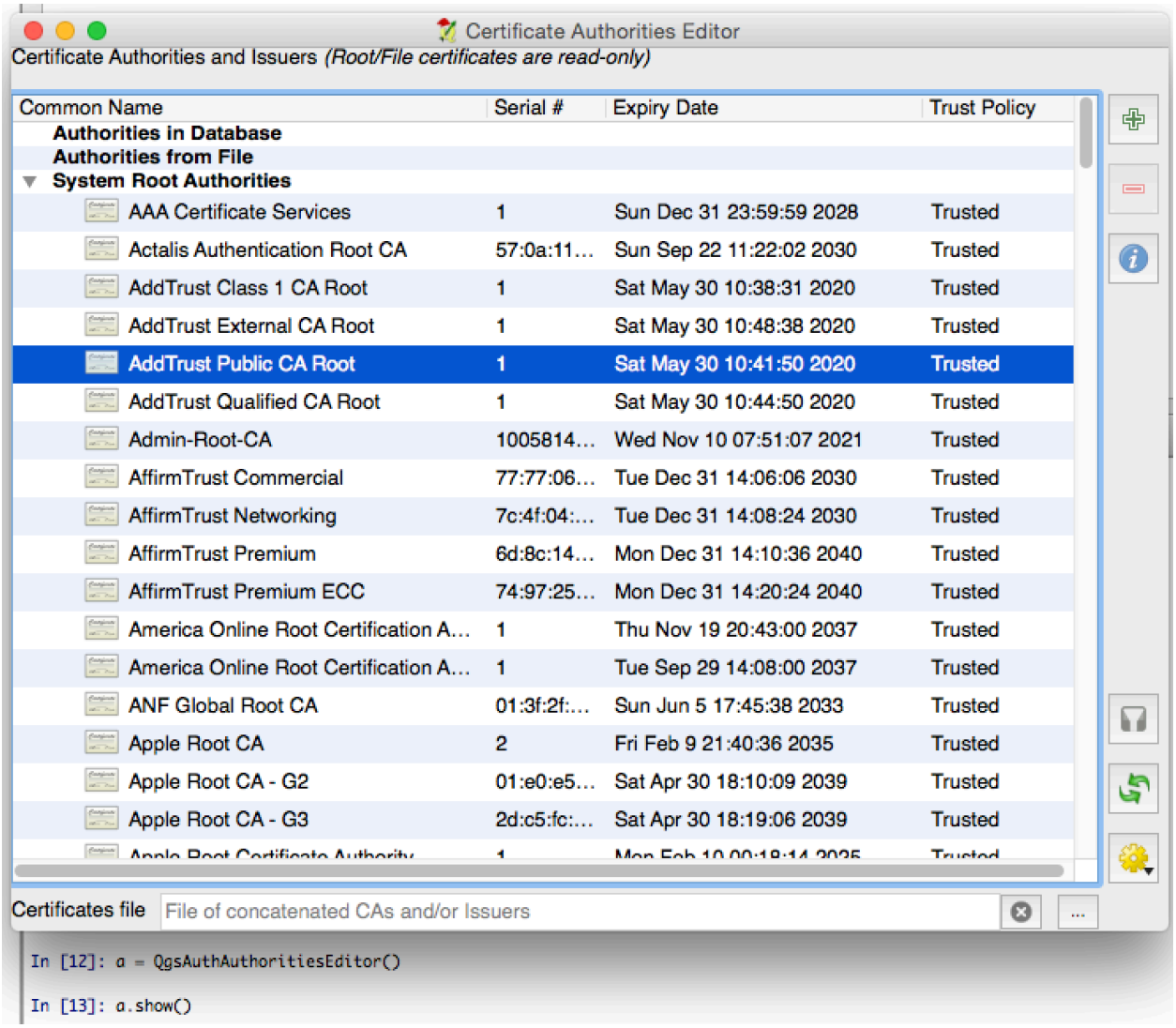
다음 코드 조각에서와 같이 사용하면 됩니다:

```
# 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()
```

관련 테스트 에서 통합된 예시를 찾아볼 수 있습니다.

14.5.3 기관 편집기 GUI

`QgsAuthAuthoritiesEditor` 클래스가 기관만을 관리하는 데 사용되는 GUI 를 관리합니다.



다음 코드 조각에서와 같이 사용하면 됩니다:

```
# 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()
```

CHAPTER 15

태스크 - 배경에서 무거운 작업 하기

> **힌트**
>
> PyQGIS 콘솔을 사용하지 않는 경우 이 페이지에 있는 코드 조각들을 다음과 같이 가져와야 합니다:

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

15.1 소개

무거운 공간 처리 작업이 진행 중일 때, 스레드들을 사용한 배경 프로세스가 즉각 반응하는 사용자 인터페이스를 유지하는 한 방법일 수 있습니다. QGIS 에서는 스레드 작업을 태스크 (task) 를 사용해서 달성할 수 있습니다.

태스크 (`QgsTask` 클래스) 란 배경에서 수행될 코드를 위한 컨테이너이며, 태스크 관리자 (`QgsTaskManager` 클래스) 는 태스크의 실행을 제어하는 데 쓰입니다. 이 클래스들은 신호 전달, 진행률 보고 및 배경 프로세스의 상태에 접근하기 위한 메커니즘을 제공해서 QGIS 에서의 배경 프로세스 작업을 단순화시켜줍니다. 태스크는 하위 태스크를 사용해서 그룹화할 수 있습니다.

일반적으로 (`QgsApplication.taskManager()` 메소드로 호출할 수 있는) 전체 수준 태스크 관리자를 사용합니다. 다시 말해 태스크 관리자가 여러분의 태스크만 제어하는 것이 아닐 수도 있다는 뜻입니다.

몇 가지 방법으로 QGIS 태스크를 생성할 수 있습니다:

- `QgsTask` 클래스를 확장시켜 사용자의 태스크를 생성합니다:

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

- 함수로부터 태스크를 생성합니다:

```
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)
```

- 공간 처리 알고리즘으로부터 태스크를 생성합니다:

```
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)
```

경고

어떤 배경 태스크도 (생성 방법에 상관없이) 주 스레드에 살아 있는, 예를 들면 QgsVectorLayer, QgsProject 에 접근하거나 또는 새 위젯을 생성하거나 기존 위젯과 쌍방향 작업을 하는 것과 같은 GUI 기반 작업을 수행하는 QObject 를 절대로 사용해서는 안 됩니다. Qt 위젯은 주 스레드에서만 접근 또는 수정해야만 합니다. 태스크에 쓰이는 데이터는 태스크를 시작하기 전에 반드시 복사해야만 합니다. 배경 스레드에서 이들을 사용하려 시도하는 경우 충돌이 발생할 것입니다.

또한 `context` 와 `feedback` 클래스들이 적어도 이들을 사용하는 태스크만큼은 살아 있도록 항상 확인하십시오. 태스크 완료 시 *QgsTaskManager* 가 태스크가 예약되었던 *context* 와 *feedback* 에 접근하지 못하는 경우 QGIS 가 중단될 것입니다.

참고

`QgsProcessingContext` 클래스를 호출한 직후 `setProject()` 메소드를 호출하는 것이 흔한 패턴입니다. 이렇게 하면 태스크는 물론 태스크의 콜백 함수가 프로젝트 전체 설정 가운데 대부분을 사용할 수 있습니다. 콜백 함수에서 공간 레이어를 작업하는 경우 특히 유용합니다.

`QgsTask` 클래스의 `addSubTask()` 함수를 사용하면 태스크들 사이의 의존성을 설명할 수 있습니다. 의존성을 선언하는 경우, 태스크 관리자가 이런 의존성들을 실행하는 방법을 자동으로 결정할 것입니다. 의존성들을 가능한 한 빨리 만족시키기 위해 의존성들을 병렬로 실행할 것입니다. 또다른 태스크가 의존하고 있는 태스크를 취소하는 경우, 의존하는 태스크도 취소될 것입니다. 순환 의존성은 교착 상태 (deadlock) 를 발생시킬 수 있기 때문에 조심해야 합니다.

태스크가 사용할 수 있는 레이어에 의존하는 경우, `QgsTask` 클래스의 `setDependentLayers()` 함수를

사용해서 이를 선언할 수 있습니다. 태스크가 의존하는 레이어를 사용할 수 없는 경우 태스크가 취소될 것입니다.

태스크를 생성하고 나면 태스크 관리자의 `addTask()` 함수를 사용해서 실행을 예약할 수 있습니다. 관리자에 태스크를 추가하면 자동적으로 관리자가 해당 태스크를 소유하게 되며, 태스크를 실행한 후 관리자가 태스크를 정리하고 삭제할 것입니다. 태스크의 예약은 태스크 우선 순위에 영향을 받는데, `addTask()` 메소드에서 이 우선 순위를 설정합니다.

`QgsTask` 및 `QgsTaskManager` 클래스의 신호와 함수를 통해 태스크의 상태를 모니터링할 수 있습니다.

15.2 예제

15.2.1 QgsTask 클래스 확장하기

다음 예시에서는 `RandomIntegerSumTask` 가 `QgsTask` 클래스를 확장시키고 지정한 시간 동안 0 에서 500 사이의 정수 100 개를 랜덤하게 생성할 것입니다. 이때 랜덤한 숫자가 42 일 경우, 태스크를 중단하고 예외를 발생시킵니다. 두 가지 유형의 의존성을 보여주는 (하위 태스크를 가진) `RandomIntegerSumTask` 의 인스턴스를 몇 개 생성해서 태스크 관리자에 추가합니다.

```
import random
from time import sleep

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

MESSAGE_CATEGORY = 'RandomIntegerSumTask'

class RandomIntegerSumTask(QgsTask):
    """This shows how to subclass QgsTask"""

    def __init__(self, description, duration):
        super().__init__(description, QgsTask.CanCancel)
        self.duration = duration
        self.total = 0
        self.iterations = 0
        self.exception = None

    def run(self):
        """Here you implement your heavy lifting.
        Should periodically test for isCanceled() to gracefully
        abort.
        This method MUST return True or False.
        Raising exceptions will crash QGIS, so we handle them
        internally and raise them in self.finished
        """
        QgsMessageLog.logMessage('Started task "{}"'.format(
                                    self.description()),
                                MESSAGE_CATEGORY, Qgis.Info)
        wait_time = self.duration / 100
        for i in range(100):
            sleep(wait_time)
            # use setProgress to report progress
            self.setProgress(i)
            arandominteger = random.randint(0, 500)
            self.total += arandominteger
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
            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:
            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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
shorttask = RandomIntegerSumTask('waste cpu short', 10)
minitask = RandomIntegerSumTask('waste cpu mini', 5)
shortsubtask = RandomIntegerSumTask('waste cpu subtask short', 5)
longsubtask = RandomIntegerSumTask('waste cpu subtask long', 10)
shortestsubtask = RandomIntegerSumTask('waste cpu subtask shortest', 4)

# Add a subtask (shortsubtask) to shorttask that must run after
# minitask and longtask has finished
shorttask.addSubTask(shortsubtask, [minitask, longtask])
# Add a subtask (longsubtask) to longtask that must be run
# before the parent task
longtask.addSubTask(longsubtask, [], QgsTask.ParentDependsOnSubTask)
# Add a subtask (shortestsubtask) to longtask
longtask.addSubTask(shortestsubtask)

QgsApplication.taskManager().addTask(longtask)
QgsApplication.taskManager().addTask(shorttask)
QgsApplication.taskManager().addTask(minitask)
```

```
RandomIntegerSumTask(0): Started task "waste cpu subtask shortest"
RandomIntegerSumTask(0): Started task "waste cpu short"
RandomIntegerSumTask(0): Started task "waste cpu mini"
RandomIntegerSumTask(0): Started task "waste cpu subtask long"
RandomIntegerSumTask(3): Task "waste cpu subtask shortest" completed
RandomTotal: 25452 (with 100 iterations)
RandomIntegerSumTask(3): Task "waste cpu mini" completed
RandomTotal: 23810 (with 100 iterations)
RandomIntegerSumTask(3): Task "waste cpu subtask long" completed
RandomTotal: 26308 (with 100 iterations)
RandomIntegerSumTask(0): Started task "waste cpu long"
RandomIntegerSumTask(3): Task "waste cpu long" completed
RandomTotal: 22534 (with 100 iterations)
```

15.2.2 함수로부터 나온 태스크

함수로부터 (이 예시에서는 `doSomething` 으로부터) 태스크를 생성합니다. 이 함수의 첫 번째 파라미터가 함수를 위한 `QgsTask` 클래스를 담을 것입니다. `on_finished` 는 태스크 완료 시 호출할 함수를 지정하는 중요한 (이름이 있는) 파라미터입니다. 이 예시의 `doSomething` 함수는 추가적인 이름이 있는 파라미터 `wait_time` 을 가지고 있습니다.

```
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()),
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                                MESSAGE_CATEGORY, Qgis.Info)
    wait_time = wait_time / 100
    total = 0
    iterations = 0
    for i in range(100):
        sleep(wait_time)
        # use task.setProgress to report progress
        task.setProgress(i)
        arandominteger = random.randint(0, 500)
        total += arandominteger
        iterations += 1
        # check task.isCanceled() to handle cancellation
        if task.isCanceled():
            stopped(task)
            return None
        # raise an exception to abort the task
        if arandominteger == 42:
            raise Exception('bad value!')
    return {'total': total, 'iterations': iterations,
            'task': task.description()}

def stopped(task):
    QgsMessageLog.logMessage(
        'Task "{name}" was canceled'.format(
            name=task.description()),
        MESSAGE_CATEGORY, Qgis.Info)

def completed(exception, result=None):
    """This is called when doSomething is finished.
    Exception is not None if doSomething raises an exception.
    result is the return value of doSomething."""
    if exception is None:
        if result is None:
            QgsMessageLog.logMessage(
                'Completed with no exception and no result '\
                '(probably manually canceled by the user)',
                MESSAGE_CATEGORY, Qgis.Warning)
        else:
            QgsMessageLog.logMessage(
                'Task {name} completed\n'
                'Total: {total} ( with {iterations} '
                'iterations)'.format(
                    name=result['task'],
                    total=result['total'],
                    iterations=result['iterations']),
                MESSAGE_CATEGORY, Qgis.Info)
    else:
        QgsMessageLog.logMessage("Exception: {}".format(exception),
                                 MESSAGE_CATEGORY, Qgis.Critical)
        raise exception

# Create a few tasks
task1 = QgsTask.fromFunction('Waste cpu 1', doSomething,
                             on_finished=completed, wait_time=4)
task2 = QgsTask.fromFunction('Waste cpu 2', doSomething,
                             on_finished=completed, wait_time=3)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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 공간 처리 알고리즘으로부터 나온 태스크

지정한 범위 안에 포인트 50,000 개를 qgis:randompointsinextent 알고리즘을 사용해서 랜덤하게 생성하는 태스크를 생성합니다. 결과물은 프로젝트에 안전한 방식으로 추가됩니다.

```
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`,
# 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'
}
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
task = QgsProcessingAlgRunnerTask(alg, params, context, feedback)
task.executed.connect(partial(task_finished, context))
QgsApplication.taskManager().addTask(task)
```

https://www.opengis.ch/2018/06/22/threads-in-pyqgis3/ 도 참조하세요.

CHAPTER 16

파이썬 플러그인 개발하기

16.1 파이썬 플러그인 구성하기

플러그인을 생성하기 위한 주요 단계는 다음과 같습니다:

1. 아이디어: 새로운 QGIS 플러그인으로 무엇을 하고 싶은지에 대한 아이디어를 떠올립니다.
2. 준비 *(setup)*: 여러분의 플러그인을 위한 파일들을 생성합니다. 플러그인 유형에 따라, 몇몇 파일은 필수인 반면 몇몇 파일은 선택적입니다.
3. 개발: 알맞은 파일에코드를 작성합니다.
4. 문서화: 플러그인 문서를 작성합니다.
5. 선택적으로 번역: 여러분의 플러그인을 다른 언어로번역합니다.
6. 테스트: 모든 것이 괜찮은지 확인하기 위해여러분의 플러그인을 다시 불러옵니다.
7. 공개 *(publish)*: QGIS 저장소에 여러분의 플러그인을 공개하거나, 또는 개인적인"GIS 무기"의"병기고"로써 여러분 자신의 저장소를 만드십시오.

16.1.1 시작하기

새 플러그인을 작성하기 전에, 공식 파이썬 플러그인 저장소 를 살펴보십시오. 기존 플러그인들의 소스 코드를 연구하면 프로그래밍에 대해 더 많은 것을 배울 수 있습니다. 비슷한 플러그인이 이미 존재하는지 찾아볼 수도 있고, 존재한다면 기존 플러그인을 확장하거나 또는 적어도 기존 플러그인을 바탕으로 여러분 자신의 플러그인을 작성할 수도 있습니다.

플러그인 파일 구조 준비하기

새 플러그인을 시작하려면, 필요한 플러그인 파일들을 준비해야 합니다.

플러그인 준비를 도와줄 수 있는 플러그인 템플릿 리소스가 2 개 있습니다:

- 교육 목적인 경우 또는 미니멀리스트 접근법이 바람직할 때마다, 최소 플러그인 템플릿 이 무결한 QGIS 파이썬 플러그인을 생성하는 데 필요한 기본 (최소한의) 파일들을 제공합니다.
- 좀 더 완전한 기능 플러그인 템플릿이 필요한 경우, 플러그인 작성자 가 현지화 (번역) 및 테스트 작업 같은 기능들을 포함하는 서로 다른 여러 플러그인 유형들을 위한 템플릿들을 생성할 수 있습니다.

전형적인 플러그인 디렉터리는 다음 파일들을 포함합니다:

- `metadata.txt` - 필수 - 플러그인 웹사이트와 플러그인 인프라스트럭처에서 사용하는 일반 정보, 버전, 이름, 그리고 몇몇 기타 메타데이터를 담고 있습니다.
- `__init__.py` - 필수 - 플러그인의 시작점입니다. 이 파일은 `classFactory()` 메소드를 가지고 있어야 하며, 다른 초기 설정 코드를 가지고 있을 수도 있습니다.
- `mainPlugin.py` - 핵심 코드 - 플러그인의 주요 작업 코드입니다. 플러그인과 주요 코드의 액션에 대한 모든 정보를 담고 있습니다.
- `form.ui` - 사용자 정의 *GUI* 를 가진 플러그인 용 - Qt 설계자가 생성한 GUI 입니다.
- `form.py` - 컴파일된 *GUI* - 앞에서 설명한 `form.ui` 를 파이썬으로 번역한 파일입니다.
- `resources.qrc` - 선택적 - Qt 설계자가 생성한 `.xml` 문서입니다. GUI 양식에 쓰이는 리소스들을 가리키는 상대 경로를 담고 있습니다.
- `resources.py` - 컴파일된 리소스, 선택적 - 앞에서 설명한 `resources.qrc` 를 파이썬으로 번역한 파일입니다.
- `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.

경고

공식 파이썬 플러그인 저장소 에 플러그인을 업로드할 계획이 있다면 여러분의 플러그인이 플러그인무결성 검증 에 필수적인 몇몇 추가 규칙들을 따르는지 반드시 확인해야만 합니다.

16.1.2 플러그인 코드 작성하기

다음은 앞에서 소개한 파일들 각각에 어떤 내용을 추가해야 하는지를 설명합니다.

metadata.txt

먼저, 플러그인 관리자는 플러그인의 이름, 설명 등등과 같은 플러그인에 대한 몇몇 기본 정보를 검색해야 합니다. `metadata.txt` 파일에 이 정보들을 저장합니다.

참고

모든 메타데이터는 반드시 UTF-8 으로 인코딩되어 있어야만 합니다.

메타데이터 이름	필수 여부	설명
name	O	플러그인의 이름을 담고 있는 짧은 문자열
qgisMinimumVersion	O	최저 QGIS 버전의 닷 데시멀 노테이션
qgisMaximumVersion	X	최고 QGIS 버전의 닷 데시멀 노테이션
description	O	플러그인을 설명하는 짧은 텍스트, HTML 은 허용되지 않습니다
about	O	플러그인을 자세히 설명하는 좀 더 긴 텍스트, HTML 은 허용되지 않습니다
version	O	버전의 닷 데시멀 노테이션을 담고 있는 짧은 문자열
author	O	개발자 이름
email	O	개발자의 이메일 주소, 웹사이트에서는 로그인한 사용자에게만 보이지만 플러그인을 설치했다면 플러그인 관리자에서 볼 수 있습니다
changelog	X	문자열, 여러 줄일 수 있지만 HTML 은 허용되지 않습니다
experimental	X	불 (boolean) 플래그, `True` 또는 `False` - 이 버전이 실험적 버전인 경우 `True`
deprecated	X	불 (boolean) 플래그, `True` 또는 `False` - 업로드된 버전만이 아니라 전체 플러그인에 적용됩니다
tags	X	쉼표로 구분된 목록, 개별 태그 안에 공백이 허용됩니다
homepage	X	사용자 플러그인의 홈페이지를 가리키는 무결한 URL
repository	O	소스 코드 저장소를 가리키는 무결한 URL
tracker	X	티켓 및 버그 보고서를 가리키는 무결한 URL
icon	X	웹 친화적인 이미지 (PNG, JPEG) 의 파일 이름 또는 (플러그인 압축 패키지의 기반 폴더를 기준으로 하는) 상대 경로
category	X	`Raster`, `Vector`, `Database`, `Mesh` 및 `Web` 가운데 하나
plugin_dependenci	X	PIP 처럼 설치할 다른 플러그인들을 쉼표로 구분한 목록, 플러그인들의 메타데이터의 이름 필드에서 나오는 플러그인 이름들을 사용합니다
server	X	불 (boolean) 플래그, `True` 또는 `False` - 플러그인이 서버 인터페이스를 가지고 있는지 여부를 결정합니다
hasProcessingProvider	X	불 (boolean) 플래그, `True` 또는 `False` - 플러그인이 공간 처리 알고리즘을 제공하는지 여부를 결정합니다

플러그인은 기본적으로 *Plugins* 메뉴에 배치되지만 (다음 부분에서 여러분의 플러그인에 대한 메뉴 항목을 추가하는 방법을 배울 것입니다) *Raster*, *Vector*, *Database*, *Mesh*, 그리고 *Web* 메뉴에도 배치할 수 있습니다.

이를 지정하기 위한 해당"category"메타데이터 항목이 존재하기 때문에, 이에 맞춰 플러그인을 분류할 수 있습니다. 이 메타데이터 항목을 사용자를 위한 도움말처럼 사용해서 어디에서 (어떤 메뉴에서) 플러그인을 찾을 수 있는지 알려줄 수 있습니다. "category"에 사용할 수 있는 값은 Vector, Raster, Database, Mesh 또는 Web 가운데 하나입니다. 예를 들어 *Raster* 메뉴에서 여러분의 플러그인을 사용할 수 있게 하려면, `metadata.txt` 에 이를 추가하십시오.

```
category=Raster
```

참고

"qgisMaximumVersion"이 비어 있는 경우, 공식 파이썬 플러그인 저장소 에 업로드할 때 자동적으로 주요 버전에 `.99` 를 더한 값으로 설정될 것입니다.

이 `metadata.txt` 의 예시:

```
; the next section is mandatory

[general]
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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

; 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

파이썬의 가져오기 시스템이 이 파일을 필요로 합니다. 또 QGIS 도 이 파일이 `classFactory()` 함수를 담고 있을 것을 요구합니다. 이 함수는 QGIS 에 플러그인을 불러왔을 때 호출되어, `QgisInterface` 클래스의 인스턴스를 받아서 `mainplugin.py` 파일로부터 플러그인의 클래스 객체를 반환해야만 합니다—이 경우 `TestPlugin` 이라는 플러그인입니다. (다음을 참조하세요.) `__init__.py` 의 내용은 다음과 같이 보여야 합니다:

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

# any other initialisation needed
```

mainPlugin.py

이 파일에서 마법이 벌어지는데, 이 마법은 다음과 같이 보일 것입니다: (`mainPlugin.py` 파일의 예시)

```
from qgis.PyQt.QtGui import *
from qgis.PyQt.QtWidgets import *

# initialize Qt resources from file resources.py
from . import resources

class TestPlugin:

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

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

    # add toolbar button and menu item
    self.iface.addToolBarIcon(self.action)
    self.iface.addPluginToMenu("&Test plugins", self.action)

    # connect to signal renderComplete which is emitted when canvas
    # rendering is done
    self.iface.mapCanvas().renderComplete.connect(self.renderTest)

  def unload(self):
    # remove the plugin menu item and icon
    self.iface.removePluginMenu("&Test plugins", self.action)
    self.iface.removeToolBarIcon(self.action)

    # disconnect form signal of the canvas
    self.iface.mapCanvas().renderComplete.disconnect(self.renderTest)

  def run(self):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    # 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!")
```

주 플러그인 소스 파일 (예: `mainPlugin.py`) 에 반드시 존재해야만 하는 플러그인 함수는 다음과 같습니다:

- `__init__`: QGIS 인터페이스에 접근할 수 있게 해줍니다.
- `initGui()`: 플러그인을 불러왔을 때 호출됩니다.
- `unload()`: 플러그인 불러오기를 해제했을 때 호출됩니다.

앞의 예시에서 `addPluginToMenu()` 메소드를 사용했습니다. 이 메소드는 *Plugins* 메뉴에 해당 메뉴 액션을 추가할 것입니다. 다른 메뉴에 액션을 추가하는 대체 메소드들이 존재합니다. 다음은 대체 메소드 목록입니다:

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

이 대체 메소드들은 모두 `addPluginToMenu()` 메소드와 동일한 문법 (syntax) 을 사용합니다.

플러그인 항목들을 정리하는 방식의 일관성을 유지하기 위해 이 사전 정의 메소드들 가운데 하나에 여러분의 플러그인 메뉴를 추가할 것을 권장합니다. 하지만, 다음 예시에서처럼 여러분이 메뉴 바에 사용자 정의 메뉴 그룹을 직접 추가할 수도 있습니다:

```
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()
```

여러분의 플러그인에 `QAction` 및 `QMenu` 의 `objectName` 을 특정 이름으로 설정해서 사용자 정의할 수 있도록 만드는 일을 잊지 마십시오.

사용자 정의 메뉴에 도움말 (help) 및 정보 (about) 액션도 추가할 수 있지만, 이 액션들을 사용할 수 있게 해주는 편리한 장소는 QGIS 의 *Help* ▯ *Plugins* 메뉴입니다. `pluginHelpMenu()` 메소드를 사용하면 됩니다.

```
def initGui(self):

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

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

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

def unload(self):

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

실제 플러그인을 작업하는 경우 또다른 (작업) 디렉터리에 플러그인을 작성하고, UI 와 리소스 파일을 생성할 `makefile` 을 생성해서 QGIS 설치본에 플러그인을 설치하는 편이 좋습니다.

16.1.3 플러그인 문서화하기

플러그인에 대한 문서는 HTML 도움말 파일로 작성할 수 있습니다. `qgis.utils` 모듈이 다른 QGIS 도움말과 동일한 방식으로 도움말 파일 탐색기를 여는 `showPluginHelp()` 함수를 제공합니다.

`showPluginHelp()` 함수는 호출한 모듈과 동일한 디렉터리에서 도움말 파일을 검색합니다. 이 함수는 `index-ll_cc.html`, `index-ll.html`, `index-en.html`, `index-en_us.html`, 그리고 `index.html` 순서대로 파일을 검색해서 첫 번째로 찾은 파일을 표시합니다. 이때 `ll_cc` 는 QGIS 로케일로, 플러그인에 문서를 여러 언어로 번역한 파일들을 포함시킬 수 있게 해줍니다.

또 `showPluginHelp()` 함수는 도움말이 표시될 특정 플러그인을 식별하는 packageName 파라미터, 검색 중인 파일들의 이름에서"index"를 대체할 수 있는 filename 파라미터, 그리고 문서 안에서 탐색기가 배치될 위치를 결정하는 HTML 앵커 태그의 이름인 section 파라미터도 받을 수 있습니다.

16.1.4 플러그인 번역하기

몇 단계만 거치면 여러분의 컴퓨터의 로케일 설정에 따라 플러그인을 서로 다른 언어들로 불러오게 만들 플러그인 현지화를 위한 환경을 준비할 수 있습니다.

소프트웨어 요구 사항

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
```

파일과 디렉터리

플러그인을 생성했을 때 주 플러그인 디렉터리 안에 `i18n` 폴더가 있을 것입니다.

모든 번역 파일은 이 디렉터리 안에 있어야 합니다.

.pro 파일

먼저 **Qt Linguist** 가 관리할 수 있는 프로젝트 파일인 `.pro` 파일을 생성해야 합니다.

이 `.pro` 파일에 번역하고자 하는 모든 파일과 양식을 지정해야 합니다. 이 파일은 현지화 파일들과 변수들을 준비하는 데 쓰입니다. 다음은예제 플러그인 의 구성과 일치하는 프로젝트 파일의 예시입니다:

```
FORMS = ../form.ui
SOURCES = ../your_plugin.py
TRANSLATIONS = your_plugin_it.ts
```

여러분의 플러그인이 좀 더 복잡한 구성을 따를 수도 있고, 파일 여러 개로 배포될 수도 있습니다. 이런 경우, `.pro` 파일을 읽고 번역할 수 있는 문자열을 업데이트하는 데 사용하는 `pylupdate5` 프로그램이 와일드카드 문자들을 지원하지 않기 때문에 `.pro` 파일에 모든 파일을 명확하게 배치해야 한다는 사실을 기억하십시오. 이때 프로젝트 파일은 다음과 같이 보일 수도 있습니다:

```
FORMS = ../ui/about.ui ../ui/feedback.ui \
        ../ui/main_dialog.ui
SOURCES = ../your_plugin.py ../computation.py \
          ../utils.py
```

이에 더해, `your_plugin.py` 파일이 QGIS 툴바에서 플러그인의 모든 메뉴와 하위 메뉴를 호출하는 파일이기 때문에, 이들도 전부 번역해야 합니다.

마지막으로 *TRANSLATIONS* 변수를 사용해서 원하는 번역 언어를 지정할 수 있습니다.

> **경고**
>
> `.ts` 파일의 이름을 `your_plugin_` + `language` + `.ts` 형식으로 붙여야 한다는 사실을 기억하십시오. 그러지 않으면 언어를 불러오지 못할 것입니다! 언어의 2 문자 단축 이름을 (이탈리아어의 경우 **it**, 독일어의 경우 **de**, …) 사용하십시오.

.ts 파일

`.pro` 파일을 생성하고 나면 플러그인의 언어 (들) 를 위한 `.ts` 파일 (들) 을 생성할 준비가 된 것입니다.

터미널을 열고, `your_plugin/i18n` 디렉터리로 가서 다음을 입력하십시오:

```
pylupdate5 your_plugin.pro
```

`your_plugin_language.ts` 파일 (들) 을 볼 수 있을 것입니다.

Qt Linguist 로 `.ts` 파일을 열고 번역을 시작하십시오.

.qm 파일

플러그인 번역을 끝내고 나면 (번역이 완료되지 않은 일부 문자열의 경우 해당 문자열의 소스 언어를 사용할 것입니다) `.qm` 파일을 생성해야 합니다. (QGIS 가 사용하게 될, 컴파일된 `.ts` 파일입니다.)

터미널을 열고, `your_plugin/i18n` 디렉터리로 가서 다음을 입력하십시오:

```
lrelease your_plugin.ts
```

이제 `i18n` 디렉터리 안에서 `your_plugin.qm` 파일 (들) 을 볼 수 있을 것입니다.

Makefile 을 사용해서 번역하기

다른 방법으로는 여러분이 플러그인 작성자를 사용해서 플러그인을 생성한 경우, 파이썬 코드와 Qt 대화창으로부터 메시지를 `makefile` 을 사용해서 추출할 수 있습니다. Makefile 의 첫머리에 LOCALES 변수가 있습니다:

```
LOCALES = en
```

이 변수에 언어의 단축 이름을 추가하십시오. 예를 들어 헝가리어를 추가한다면:

```
LOCALES = en hu
```

이제 다음 명령어를 실행하면 소스로부터 (`en.ts` 파일은 물론) `hu.ts` 파일을 생성하거나 업데이트할 수 있습니다:

```
make transup
```

이 명령어를 실행하고 나면 LOCALES 변수에 설정된 모든 언어에 대한 `.ts` 파일이 업데이트됩니다. **Qt Linguist** 를 사용해서 프로그램 메시지를 번역하십시오. 번역을 완료하면 `transcompile` 로 `.qm` 파일을 생성할 수 있습니다:

```
make transcompile
```

`.ts` 파일을 여러분의 플러그인과 함께 배포해야 합니다.

플러그인 불러오기

플러그인의 번역을 보려면, QGIS 를 열어서 언어를 변경한 다음 (*Settings* ⇒ *Options* ⇒ *General*) QGIS 를 다시 시작하십시오.

여러분의 플러그인을 정확한 언어로 보게 될 것입니다.

> **경고**
>
> 플러그인에서 무언가를 (새로운 UI, 새로운 메뉴 등등) 변경하는 경우 `.ts` 와 `.qm` 파일 둘 다 업데이트 버전을 **다시 생성** 해야 하기 때문에, 앞의 명령어를 다시 실행하십시오.

16.1.5 사용자 플러그인 공유하기

QGIS 는 플러그인 저장소에 수백 개의 플러그인을 호스팅하고 있습니다. 여러분의 플러그인을 공유해보십시오! 그러면 QGIS 의 가능성이 확장되는 것은 물론 사람들이 여러분의 코드로부터 배울 수 있을 것입니다. QGIS 에서 호스팅하고 있는 모든 플러그인을 플러그인 관리자를 통해 검색하고 설치할 수 있습니다.

plugins.qgis.org 에서 관련 정보와 요구 사항을 찾아볼 수 있습니다.

16.1.6 도움말 및 꼼수

플러그인 다시 불러오기

여러분이 플러그인을 개발하는 동안 테스트를 위해 QGIS 에 플러그인을 자주 다시 불러와야 할 것입니다. 이때 **플러그인 다시 불러오기 (Plugin Reloader)** 플러그인을 사용하면 플러그인을 매우 쉽게 다시 불러올 수 있습니다. 이 플러그인은 플러그인 관리자 를 통해 찾을 수 있습니다.

qgis-plugin-ci 를 사용한 자동 패키지 작업, 배포 및 번역

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.

이 파이썬 프로그램은 CLI(명령 줄 인터페이스) 를 통해 또는 CI(지속적 통합) 에서의 액션으로 XML 플러그인 저장소 파일을 배포하고, 번역하고, 공개하고, 또는 생성할 수 있게 해줍니다.

플러그인에 접근하기

QGIS 안에서, 설치된 플러그인의 모든 클래스에 접근할 수 있습니다. 디버그 작업을 수행하는 경우에 편리합니다.

```
my_plugin = qgis.utils.plugins['My Plugin']
```

로그 메시지

log_message_panel 안에 플러그인 전용 탭이 있습니다.

리소스 파일

일부 플러그인은 리소스 파일을 사용합니다. 예를 들어 `resources.qrc` 파일은 아이콘 같은 GUI 용 리소스를 정의합니다:

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

다른 플러그인들 또는 QGIS 의 어떤 부분과도 충돌하지 않을 접두어를 사용하는 편이 좋습니다. 그러지 않으면 원하지 않는 리소스를 얻게 될 수도 있습니다. 이제 리소스를 담게 될 파이썬 파일을 **pyrcc5** 명령어를 사용해서 생성해야 합니다:

```
pyrcc5 -o resources.py resources.qrc
```

> **참고**
>
> 윈도우 환경에서 명령 프롬프트 또는 파워셸에서 **pyrcc5** 를 실행하려 하면 아마도"지정한 장치, 경로 또는 파일에 액세스할 수 없습니다. [⋯]"오류가 발생할 것입니다. 가장 쉬운 해결책은 OSGeo4W 셸을 사용하는 것일 테지만, 여러분이 PATH 환경 변수를 수정하는 데 또는 실행 파일을 명확하게 가리키는 경로를 지정하는 데 익숙하다면 그 실행 파일은 `<QGIS 설치 디렉터리>\bin\pyrcc5.exe` 입니다.

16.2 코드 조각

> **힌트**
>
> PyQGIS 콘솔을 사용하지 않는 경우 이 페이지에 있는 코드 조각들을 다음과 같이 가져와야 합니다:

```
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
```

이 장에서는 플러그인을 쉽게 개발할 수 있도록 도와주는 코드 조각 (snippet) 에 대해 설명합니다.

16.2.1 단축키로 메소드를 호출하는 방법

플러그인의 `initGui()` 함수에 다음 코드를 추가하십시오:

```
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()` 함수에 다음 코드를 추가하십시오:

```
self.iface.unregisterMainWindowAction(self.key_action)
```

CTRL+I 단축키를 누르면 해당 메소드를 호출합니다:

```
def key_action_triggered(self):
  QMessageBox.information(self.iface.mainWindow(),"Ok", "You pressed Ctrl+I")
```

사용자가 해당 액션에 단축키를 사용자 정의할 수 있게 할 수도 있습니다. 다음 코드를 추가하면 됩니다:

```
# 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 아이콘을 재사용하는 방법

QGIS 아이콘이 잘 알려져 있고 사용자에게 명확한 메시지를 전달하기 때문에, 여러분의 플러그인에 새 아이콘을 그려서 설정하는 대신 QGIS 아이콘을 재사용하고 싶을 경우가 있을 수도 있습니다. 이런 경우 `getThemeIcon()` 메소드를 사용하십시오.

예를 들어 QGIS 코드 저장소에서 사용할 수 있는 mActionFileOpen.svg 아이콘을 재사용하려면:

```
# 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)
```

QGIS 아이콘을 호출할 수 있는 또다른 메소드로는 `iconPath()` 가 있습니다. QGIS 내장 이미지 - 치트시트 에서 테마 아이콘을 호출하는 예시를 찾아보십시오.

16.2.3 옵션 대화창의 플러그인 용 인터페이스

Settings ▯ Options 에 사용자 정의 플러그인 옵션 탭을 추가할 수 있습니다. 사용자 플러그인의 옵션을 위해 특정 메인 메뉴 항목을 추가하는 것보다는 이 편이 좋은데, 모든 QGIS 응용 프로그램 설정과 플러그인 설정을 단일 위치에 모아두기 때문에 사용자가 설정을 찾고 탐색하기에 더 편하기 때문입니다.

다음 코드 조각은 플러그인의 설정에 새 비어 있는 탭만 추가할 것입니다. 여러분이 이 탭에 사용자 플러그인에 특화된 모든 옵션 및 설정을 채워 넣을 수 있습니다. 다음 클래스들을 서로 다른 파일들로 분할할 수 있습니다. 이 예제에서 우리는 주 `mainPlugin.py` 파일 안에 클래스 2 개를 추가할 것입니다.

```
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)
```

마지막으로 가져온 것들을 추가하고 `__init__` 함수를 수정합니다:

```
from qgis.PyQt.QtWidgets import QHBoxLayout
from qgis.gui import QgsOptionsWidgetFactory, QgsOptionsPageWidget


class MyPlugin:
    """QGIS Plugin Implementation."""

    def __init__(self, iface):
        """Constructor.

        :param iface: An interface instance that will be passed to this class
            which provides the hook by which you can manipulate the QGIS
            application at run time.
        :type iface: QgsInterface
        """
        # Save reference to the QGIS interface
        self.iface = iface


    def initGui(self):
        self.options_factory = MyPluginOptionsFactory()
        self.options_factory.setTitle(self.tr('My Plugin'))
        iface.registerOptionsWidgetFactory(self.options_factory)

    def unload(self):
        iface.unregisterOptionsWidgetFactory(self.options_factory)
```

> **팁**
>
> **레이어 속성 대화창에 사용자 정의 탭 추가하기**
>
> 비슷한 논리를 적용해서 `QgsMapLayerConfigWidgetFactory` 및 `QgsMapLayerConfigWidget` 클래스를 사용하면 레이어 속성 대화창에 플러그인 사용자 정의 옵션을 추가할 수 있습니다.

16.2.4 레이어 트리에 있는 레이어에 사용자 정의 위젯 내장시키기

Layers 패널에 있는 레이어 항목 옆 또는 아래에 표시되는 일반 레이어 심볼 요소 이외에도, 레이어에 자주 쓰이는 (필터링 설정, 선택, 스타일, 버튼 위젯으로 레이어 새로고침, 시간 슬라이드 바를 바탕으로 레이어 생성, 또는 라벨에 추가 레이어 정보만 보이게 하기, …) 몇몇 액션에 빠르게 접근할 수 있게 해주는 여러분의 위젯을 추가할 수 있습니다. 이 소위 **레이어 트리 내장 위젯** 은 개별 레이어에 대한 레이어 속성 대화창의 *Legend* 탭을 통해 사용할 수 있습니다.

다음 코드 조각은 범례 안에 레이어에 사용할 수 있는 레이어 스타일을 보여줘서 서로 다른 레이어 스타일들을 빠르게 전환할 수 있는 드롭 다운 메뉴를 생성합니다.

```
class LayerStyleComboBox(QComboBox):
    def __init__(self, layer):
        QComboBox.__init__(self)
        self.layer = layer
        for style_name in layer.styleManager().styles():
            self.addItem(style_name)

        idx = self.findText(layer.styleManager().currentStyle())
        if idx != -1:
          self.setCurrentIndex(idx)

        self.currentIndexChanged.connect(self.on_current_changed)

    def on_current_changed(self, index):
        self.layer.styleManager().setCurrentStyle(self.itemText(index))

class LayerStyleWidgetProvider(QgsLayerTreeEmbeddedWidgetProvider):
    def __init__(self):
        QgsLayerTreeEmbeddedWidgetProvider.__init__(self)

    def id(self):
        return "style"

    def name(self):
        return "Layer style chooser"

    def createWidget(self, layer, widgetIndex):
        return LayerStyleComboBox(layer)

    def supportsLayer(self, layer):
        return True   # any layer is fine

provider = LayerStyleWidgetProvider()
QgsGui.layerTreeEmbeddedWidgetRegistry().addProvider(provider)
```

그러면 지정한 레이어의 *Legend* 속성 탭에서, *Available widgets* 로부터 `Layer style chooser` 를 *Used widgets* 로 드래그하면 해당 위젯이 레이어 트리에 활성화됩니다. 내장 위젯은 항상 관련 레이어 노드 하위 항목의 최상단에

표시됩니다.

예를 들어 플러그인 내부에서 위젯을 사용하고 싶은 경우, 다음과 같이 추가하면 됩니다:

```
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 플러그인 작성 및 디버그 작업을 위한 IDE 설정

프로그래머마다 선호하는 IDE/텍스트 편집기가 있을 테지만, 이 장에서는 QGIS 파이썬 플러그인을 작성하고 디버그하기 위한 유명한 IDE 몇 개를 추천합니다.

16.3.1 파이썬 플러그인 작성에 유용한 플러그인들

파이썬 플러그인 작성을 편리하게 해주는 플러그인이 몇 개 있습니다. *Plugins ⸮ Manage and Install plugins…* 메뉴에서 다음을 설치하십시오:

- 플러그인 다시 불러오기 *(Plugin reloader)*: QGIS 를 다시 시작하지 않고서도 플러그인을 다시 불러와 새로운 변경 사항을 적용할 수 있게 해줍니다.
- 응급 처치 *(First Aid)*: 플러그인에서 예외가 발생한 경우 변수들을 조사할 수 있는 파이썬 콘솔과 로컬 디버그 작업자를 추가합니다.

경고

지속적으로 애를 쓰고 있지만, 이 줄 다음에 나오는 정보들이 QGIS 3 버전에 대해 업데이트되지 않았을 수도 있습니다.

16.3.2 리눅스 및 윈도우 상에서 사용자의 IDE 환경설정에 대한 첨언

리눅스 상에서, 일반적으로 해야 하는 일은 사용자의 `PYTHONPATH` 환경 변수에 QGIS 라이브러리 위치를 추가하는 것이 전부입니다. 대부분의 배포판들에서는 `~/.bashrc` 또는 `~/.bash-profile` 파일을 다음 줄을 사용해서 편집하면 됩니다. (오픈수세 텀블위드 (openSUSE Tumbleweed) 버전에서 테스트했습니다.):

```
export PYTHONPATH="$PYTHONPATH:/usr/share/qgis/python/plugins:/usr/share/qgis/python"
```

파일을 저장하고, 다음 셸 명령어를 사용해서 환경 설정을 구현하십시오:

```
source ~/.bashrc
```

윈도우 상에서는, 여러분이 QGIS 와 동일한 환경을 설정했는지 그리고 동일한 라이브러리와 해석기를 사용하고 있는지 확인해야 합니다. QGIS 의 구동 배치 (batch) 파일을 수정하는 것이 가장 빠른 방법입니다.

OSGeo4W 설치자를 사용한 경우, 여러분의 OSGeo4W 설치 위치의 `bin` 폴더에서 이 파일을 찾을 수 있습니다. `C:\OSGeo4W\bin\qgis-unstable.bat` 같은 경로를 검색해보십시오.

16.3.3 파이스크립터 IDE 를 사용한 디버그 작업 (윈도우)

파이스크립터 (Pyscripter) IDE 를 사용하려면, 다음 작업을 해야 합니다:

1. `qgis-unstable.bat` 파일을 복사한 다음 복사본의 이름을 `pyscripter.bat` 으로 바꾸십시오.
2. 편집기에서 이 파일을 연 다음, QGIS 를 구동하는 마지막 줄을 제거하십시오.
3. 파이스크립터 실행 파일을 가리키는 줄을 추가한 다음 사용할 파이썬 버전을 설정하는 명령줄 인자를 추가하십시오.
4. 또 파이스크립터가 QGIS 가 사용하는 파이썬 DLL 파일을 찾을 수 있는 폴더를 가리키는 인자를 추가하십시오. 이 폴더는 OSGeo4W 설치 위치의 `bin` 폴더에서 찾을 수 있습니다.

```
@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. 이제 이 배치 파일을 더블 클릭하면 정확한 경로를 가진 파이스크립터를 시작할 것입니다.

파이스크립터보다 더 유명한 이클립스 (Eclipse) 는 개발자들이 흔히 선택하곤 하는 IDE 입니다. 다음 항에서는 플러그인을 개발하고 테스트하기 위해 이클립스를 어떻게 환경설정해야 하는지를 설명할 것입니다.

16.3.4 이클립스 및 파이데브를 사용한 디버그 작업

설치

이클립스를 사용하기 위해서는 다음을 설치했는지 확인해야 합니다:

- Eclipse
- Aptana Studio 3 Plugin or PyDev
- QGIS 3.x
- QGIS 플러그인 **원격 디버그 (Remote Debug)** 를 설치하려 할 수도 있습니다. 현 시점에서 이 플러그인은 아직 실험적이기 때문에, 설치하기 전에 *Plugins → Manage and Install plugins··· → Options* 메뉴에서 ☑ *Experimental plugins* 옵션을 활성화시키십시오.

윈도우에서 이클립스를 사용하기 위한 환경을 준비하려면, 이클립스를 구동하는 배치 파일도 생성해서 사용해야 합니다:

1. `qgis_core.dll` 파일이 있는 폴더를 찾으십시오. 일반적으로 `C:\OSGeo4W\apps\qgis\bin` 폴더이지만, 여러분이 직접 QGIS 응용 프로그램을 컴파일한 경우 이 폴더는 빌드 폴더 아래 있는 `output/bin/RelWithDebInfo` 폴더입니다.
2. `eclipse.exe` 실행 파일의 위치를 찾으십시오.
3. 다음 스크립트를 생성한 다음 QGIS 플러그인 개발 시 이 스크립트를 사용해서 이클립스를 구동하십시오.

```
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
```

이클립스 설정하기

1. 이클립스에서 새 프로젝트를 생성하십시오. 일반 프로젝트 *(General Project)* 를 선택한 다음 나중에 실제 리소스를 링크시킬 수 있습니다. 즉 여러분이 이 프로젝트를 어디에 배치하느냐는 그렇게 중요하지 않습니다.
2. 새 프로젝트를 오른쪽 클릭한 다음 *New → Folder* 메뉴를 선택하십시오.
3. *Advanced* 를 클릭하고 *Link to alternate location (Linked Folder)* 를 선택하십시오. 여러분이 이미 디버그하고자 하는 소스를 가지고 있는 경우 해당 리소스를 선택하십시오. 그렇지 않다면, 앞에서 설명한 것처럼 폴더를 하나 생성하십시오.

이제 *Project Explorer* 뷰에 여러분의 소스 트리가 뜨고 여러분이 코드 작업을 시작할 수 있습니다. 처음부터 문법 강조 및 다른 모든 강력한 IDE 도구들을 사용할 수 있습니다.

디버그 작업자 환경설정하기

디버그 작업자를 작동하게 하려면:

1. 이클립스에서 디버그 관점으로 전환합니다. (*Window → Open Perspective → Other → Debug*)
2. *PyDev → Start Debug Server* 메뉴를 선택해서 파이데브 디버그 서버를 시작합니다.
3. 이클립스는 이제 QGIS 가 디버그 서버로 연결하기를 기다립니다. QGIS 가 디버그 서버에 연결되면 QGIS 는 이클립스가 파이썬 스크립트를 제어할 수 있게 해줄 것입니다. 이것이 바로 우리가 원격 디버그 플러그인을 설치한 이유입니다. 따라서 QGIS 를 아직 시작하지 않았다면 QGIS 를 시작한 다음 버그 심볼을 클릭하십시오.

이제 중단점 (breakpoint) 을 설정할 수 있기 때문에, 코드가 중단점에 도달하는 순간 실행을 멈추고 여러분의 플러그인의 현재 상태를 조사할 수 있습니다. (중단점은 다음 그림에 있는 녹색 점으로, 중단점을 설정하려는 줄 왼쪽의 하얀 공간을 더블 클릭해서 설정합니다.)

이제 디버그 콘솔이라는 매우 재미있는 물건을 사용할 수 있게 되었습니다. 디버그 콘솔로 진행하기 전에 먼저 실행이 현재 중단점에서 중단된 상태인지 확인하십시오.

1. 콘솔 뷰를 (*Window → Show view*) 여십시오. 그러면 *Debug Server* 콘솔이 열리는데, 그렇게 재미있어 보이진 않습니다. 하지만 *Open Console* 버튼을 누르면 좀 더 재미있는 파이데브 디버그 콘솔로 전환시킬 수 있습니다.
2. *Open Console* 버튼 옆에 있는 화살표를 클릭한 다음 파이데브 콘솔 *(PyDev Console)* 을 선택하십시오. 여러분에게 어떤 콘솔을 시작하고 싶은지 묻는 창이 열립니다.
3. 파이데브 디버그 콘솔 *(PyDev Debug Console)* 을 선택하십시오. 이 항목이 회색조로 비활성화되어 있고 디버그 작업자를 시작해서 무결한 프레임을 선택하라는 메시지를 보이는 경우, 원격 디버그 플러그인을 활성화했는지 그리고 현재 중단점에서 중단된 상태인지 확인하십시오.

이제 여러분이 현재 맥락 안에서 어떤 명령어든 테스트할 수 있게 해주는 쌍방향 작업 콘솔이 열렸습니다. 변수를 조작하거나 API 를 호출하거나 원하는 것은 무엇이든 할 수 있습니다.

> **팁**
>
> 조금 짜증스러운 사실은, 여러분이 명령어를 입력할 때마다 콘솔이 디버그 서버로 다시 전환된다는 점입니다. 이 습성을 중단시키려면, 디버그 서버 페이지에 있을 때 콘솔 고정 *(Pin Console)* 버튼을 클릭하면 됩니다. 적어도 현재 디버그 세션 동안에는 이 결정을 기억할 것입니다.

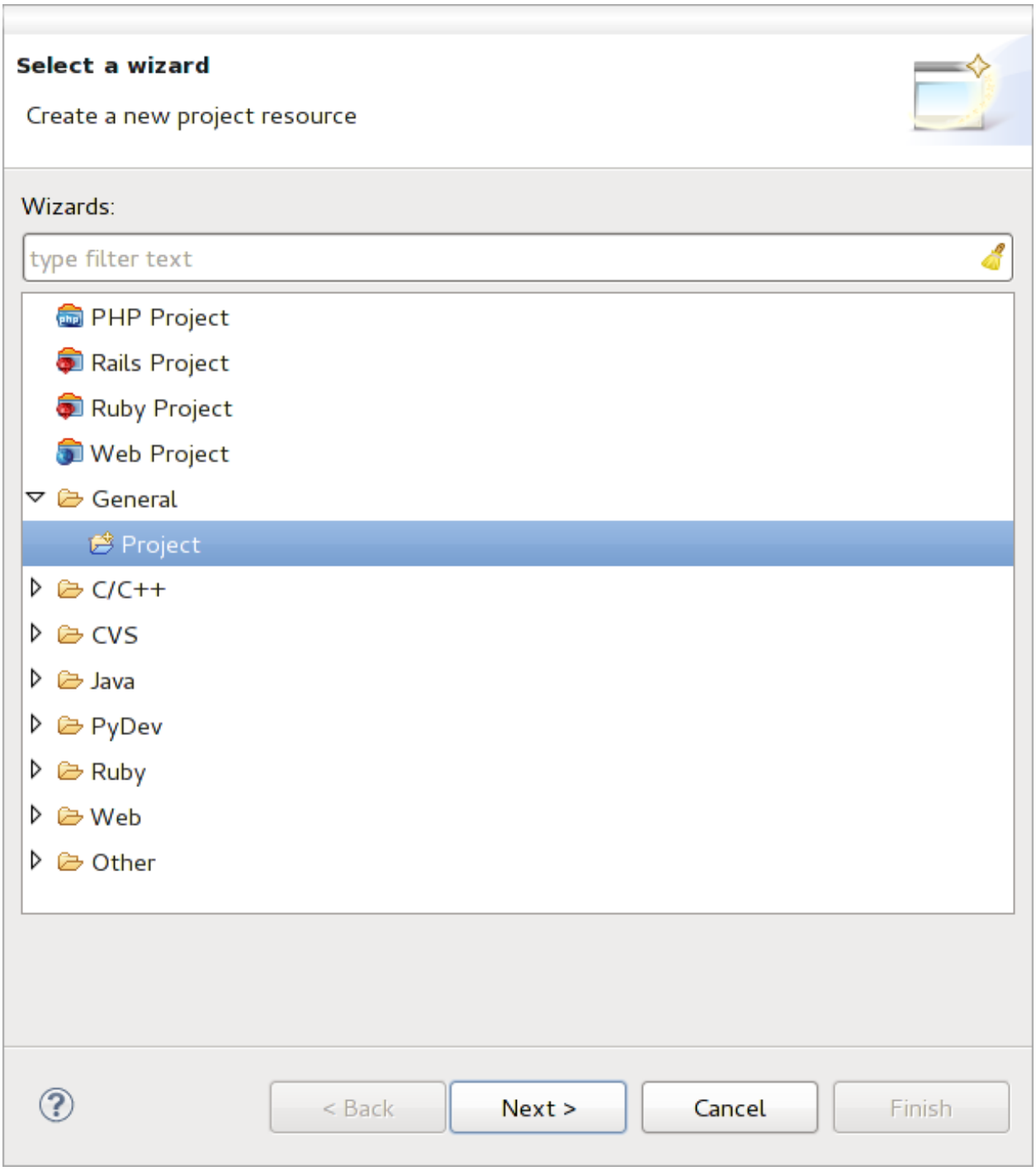


그림 16.1: 이클립스 프로젝트

```
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 )
```

그림 16.2: 중단점

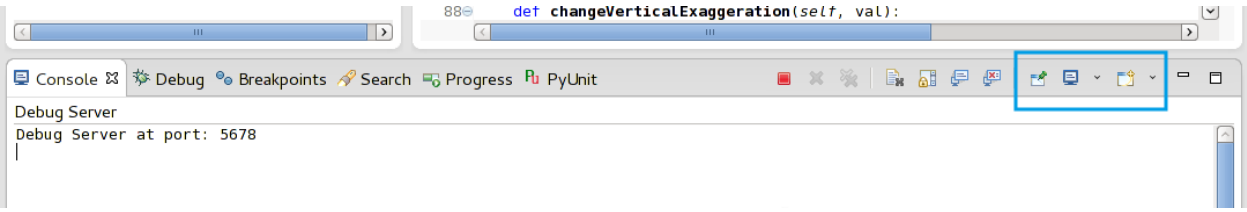


그림 16.3: 파이데브 디버그 콘솔

이클립스에 API 를 이해시키기

이클립스가 실제로 QGIS API 에 대해 알도록 해주는 매우 편리한 기능이 있습니다. 이 기능은 이클립스가 코드의 오타를 점검할 수 있게 해줍니다. 이뿐만이 아니라, 이클립스가 가져오기로부터 API 호출에 자동 완성 기능을 사용할 수 있게 해줍니다.

이클립스는 이를 위해 QGIS 라이브러리 파일들을 파싱해서 모든 정보를 가져옵니다. 여러분이 해야 할 일은 이클립스에 라이브러리의 위치를 알려주는 것뿐입니다.

1. *Window → Preferences → PyDev → Interpreter → Python* 메뉴를 클릭하십시오.

   창 윗부분에 사용자가 환경설정한 (현 시점에서는 QGIS 용 파이썬 2.7) 파이썬 해석기가 나타날 것입니다. 아랫부분에는 탭 몇 개가 보일 것입니다. 여기서 흥미로운 탭은 라이브러리 *(Libraries)* 와 강제 내장 *(Forced Builtins)* 탭입니다.

2. 먼저 라이브러리 탭을 여십시오.

3. *New Folder* 를 클릭하고 QGIS 설치본의 파이썬 폴더를 선택하십시오. 이 폴더의 위치를 모르는 경우 (플러그인 폴더가 아닙니다):

   1. QGIS 를 여십시오.
   2. 파이썬 콘솔을 시작하십시오.
   3. `qgis` 를 입력하십시오.
   4. 그리고 엔터 키를 누르십시오. 파이썬이 사용하는 QGIS 모듈과 그 경로를 출력할 것입니다.
   5. 이 경로 맨 뒤에 있는 `/qgis/__init__.pyc` 문자열을 제거한 경로가 파이썬 폴더를 가리키는 경로입니다.

4. 여기에 플러그인 폴더도 추가해야 합니다. (사용자 프로파일 폴더 아래 있는 `python/plugins` 폴더입니다.)

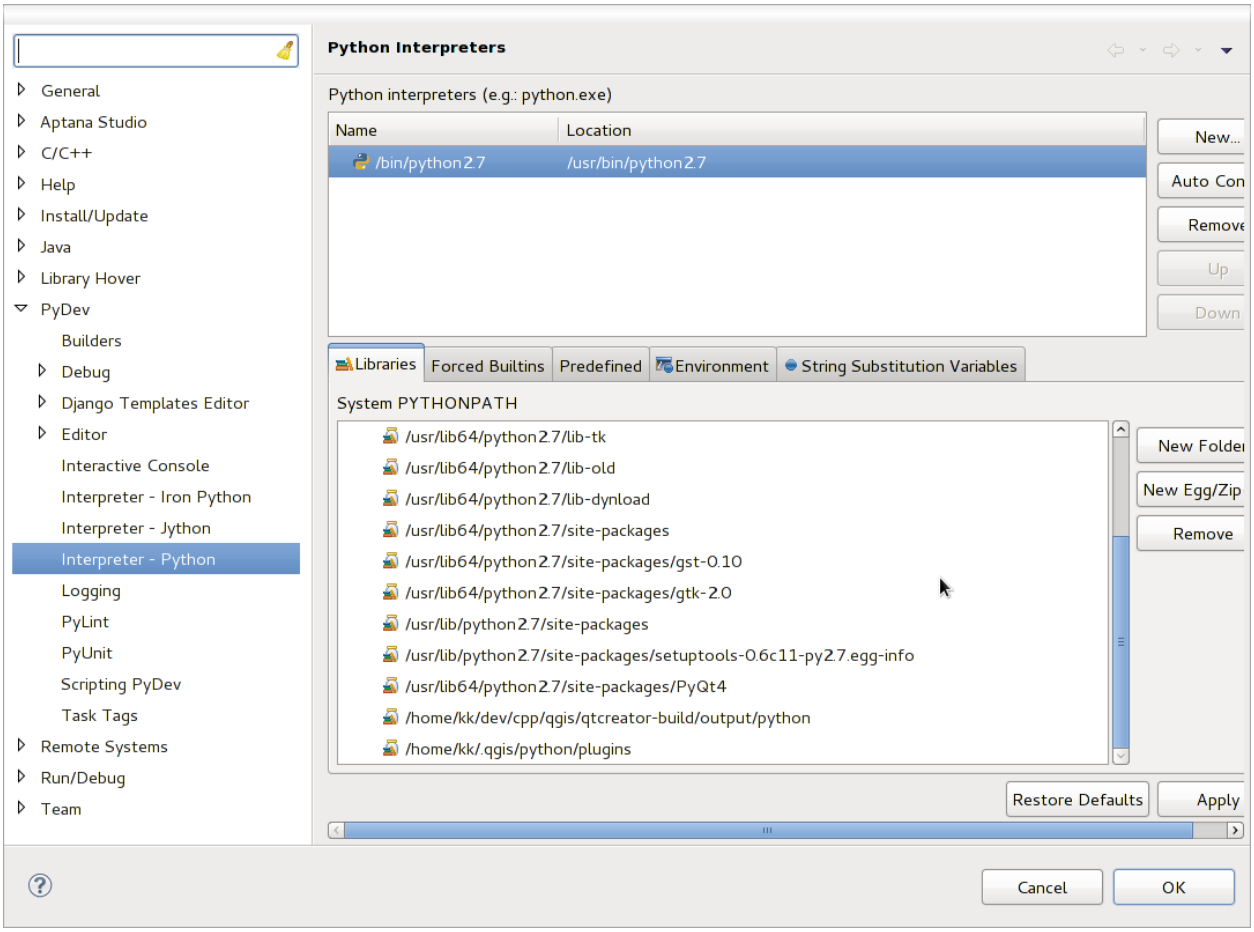


그림 16.4: 파이데브 디버그 콘솔

5. 그 다음 강제 내장 탭으로 가서 *New…*를 클릭한 다음 `qgis` 를 입력하십시오. 그러면 이클립스가 QGIS API 를 파싱할 것입니다. 이클립스가 PyQt API 에 대해서도 알고 있기를 원할 수도 있습니다. 따라서 PyQt 도 강제 내장으로 추가하십시오. PyQt 는 이미 라이브러리 탭에 있을 것입니다.
6. *OK* 를 클릭하면 끝입니다.

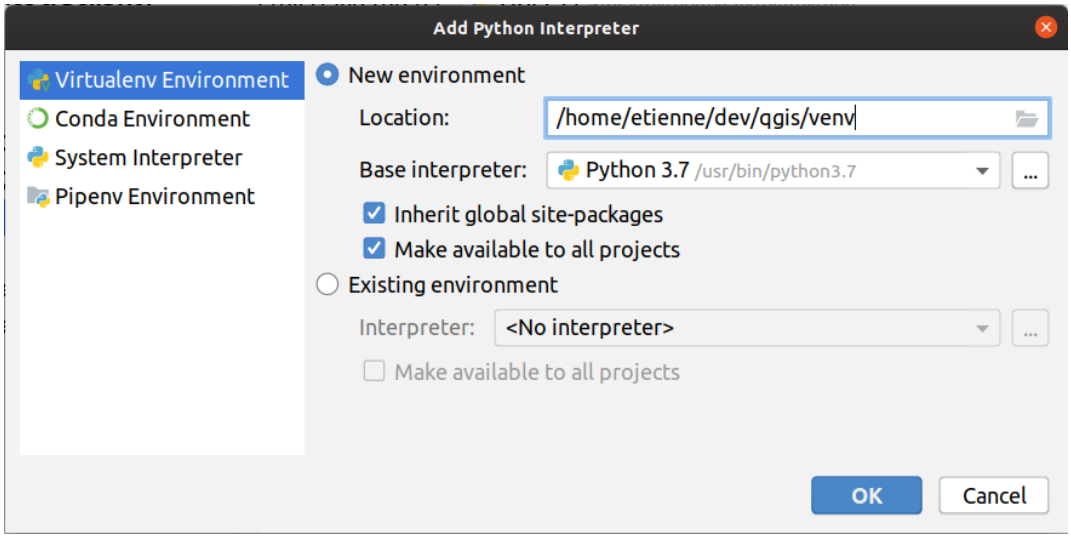
> **참고**
>
> QGIS API 가 변경될 때마다 (예를 들면 QGIS 마스터 분기를 컴파일하는 도중 SIP 파일이 변경되는 경우) 다시 이 페이지로 돌아가서 *Apply* 를 클릭하기만 하십시오. 이클립스가 모든 라이브러리를 다시 파싱할 것입니다.

16.3.5 QGIS 가 컴파일된 우분투에서 파이참을 사용한 디버그 작업

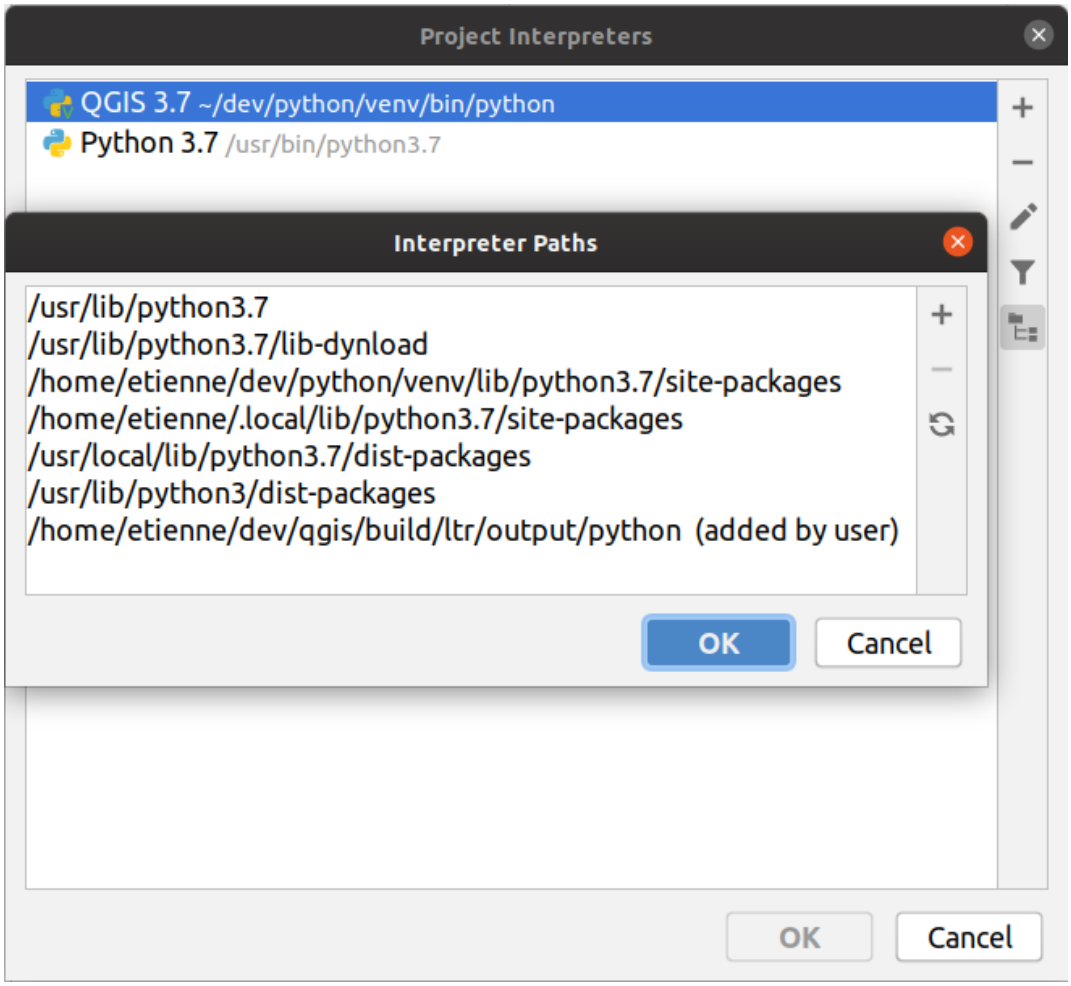
파이참 (PyCharm) 은 젯브레인스 (JetBrains) 에서 개발한 파이썬 용 IDE 입니다. 커뮤니티 에디션이라는 무료 버전과 프로페셔널 에디션이라는 유료 버전이 있습니다. https://www.jetbrains.com/pycharm/download 웹사이트에서 파이참을 다운로드할 수 있습니다.

여러분이 우분투 상에 지정한 `~/dev/qgis/build/master` 빌드 디렉터리를 사용해서 QGIS 를 컴파일했다고 가정합니다. 여러분이 직접 QGIS 를 컴파일해야만 하는 것은 아니지만, 컴파일된 QGIS 에 대해서만 테스트되었습니다. QGIS 를 직접 컴파일하지 않은 경우, 경로들을 조정해야만 합니다.

1. 파이참의 *Project Properties* 및 *Project Interpreter* 에서 `QGIS` 라는 파이썬 가상 환경을 생성할 것입니다.
2. 작은 톱니바퀴 아이콘을 클릭한 다음 *Add* 를 클릭하십시오.
3. *Virtualenv environment* 를 선택하십시오.
4. `~/dev/qgis/venv` 같은 여러분의 모든 파이썬 프로젝트의 일반 위치를 선택하십시오. 모든 플러그인에 대해 이 파이썬 해석기를 사용할 것이기 때문입니다.
5. 여러분의 시스템 상에서 사용할 수 있는 파이썬 3 버전 기반 해석기를 선택하고 다음 *Inherit global site-packages* 및 *Make available to all projects* 옵션 2 개를 체크하십시오.



1. *OK* 를 클릭한 다음, 다시 작은 톱니바퀴 아이콘으로 돌아와 *Show all* 을 클릭하십시오.
2. 새 창이 열리면, 새 `QGIS` 해석기를 선택한 다음 *Show paths for the selected interpreter* 수직 메뉴의 마지막 아이콘을 클릭하십시오.
3. 마지막으로, 목록에 다음 `~/dev/qgis/build/master/output/python` 절대 경로를 추가하십시오.


Project Interpreters
QGIS 3.7 ~/dev/python/venv/bin/python
Python 3.7 /usr/bin/python3.7
Interpreter Paths
/usr/lib/python3.7
/usr/lib/python3.7/lib-dynload
/home/etienne/dev/python/venv/lib/python3.7/site-packages
/home/etienne/.local/lib/python3.7/site-packages
/usr/local/lib/python3.7/dist-packages
/usr/lib/python3/dist-packages
/home/etienne/dev/qgis/build/ltr/output/python (added by user)
OK
Cancel
OK
Cancel

1. 파이참을 다시 시작하면 여러분의 모든 플러그인에 대해 이 새로운 파이썬 가상 환경을 사용할 수 있습니다.

파이참은 QGIS API 는 물론 `from qgis.PyQt.QtCore import QDir` 같이 QGIS 가 제공하는 Qt 를 사용하는 경우 PyQt API 도 인식하게 될 것입니다.

파이참의 프로페셔널 에디션에서는 원격 디버그 작업도 잘 작동합니다. 커뮤니티 에디션의 경우, 원격 디버그를 사용할 수 없습니다. 로컬 디버그 작업자에만 접근할 수 있는데, QGIS 자체에서가 아니라 파이참 내부에서 (스크립트로써 또는 unittest 로) 코드를 실행해야만 한다는 뜻입니다. QGIS 안에서 실행되는 파이썬 코드의 경우, 앞에서 설명한 응급 처치 플러그인을 사용할 수도 있습니다.

16.3.6 PDB 를 사용한 디버그 작업

이클립스 또는 파이참 같은 IDE 를 사용하지 않는 경우, 다음 단계를 따라 PDB(Python Debugger) 를 사용해서 디버그할 수 있습니다.

1. 먼저 디버그하고자 하는 위치에 다음 코드를 추가하십시오:

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

2. 그리고 명령줄에서 QGIS 를 실행하십시오.

   리눅스에서:

```
$ ./Qgis
```

   맥 OS 에서:

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

3. 그러면 응용 프로그램이 중단점에 도달했을 때 콘솔에 입력할 수 있게 됩니다!

할 일:
테스트 정보 추가하기

16.4 사용자 플러그인 배포하기

여러분의 플러그인이 준비되었고 이 플러그인이 몇몇 사람에게 도움이 될 거라 생각한다면, 주저 말고공식 파이썬 플러그인 저장소 에 업로드하십시오. 해당 페이지에서 플러그인 설치자와 잘 작동하도록 플러그인을 준비하는 방법에 대한 패키지 작업 지침도 찾아볼 수 있습니다. 또는 사용자 자신의 플러그인 저장소를 준비하고 싶은 경우, 플러그인과 그 메타데이터를 목록화할 단순 XML 파일을 생성하십시오.

다음 권고 사항들에 특히 신경을 써주십시오:

16.4.1 메타데이터와 이름

- 기존 플러그인과 너무 비슷한 이름을 사용하지 말아주십시오.
- 여러분의 플러그인이 기존 플러그인과 유사한 기능을 가지고 있는 경우, 사용자가 플러그인을 설치해서 테스트할 필요 없이 어떤 플러그인을 사용해야 할지 알 수 있도록'About'필드에 차이점을 설명해주십시오.
- 플러그인 이름 자체에"plugin"을 반복하지 말아주십시오.
- 메타데이터의'Description'필드에는 한 줄 설명을 그리고'About'필드에는 좀 더 상세한 지침을 작성해주십시오.
- 코드 저장소, 버그 추적기, 그리고 홈페이지를 포함시켜주십시오. 그렇게 하면 공동 작업의 가능성을 크게 높여주며, 사용할 수 있는 웹 인프라스트럭처 (깃허브 (Github), 깃랩 (Gitlab), 비트버킷 (Bitbucket) 등등) 가운데 하나를 사용해서 공동 작업을 매우 쉽게 할 수 있습니다.
- 주의를 기울여 태그를 선택하십시오: 쓸데없는 정보를 주는 (예: 벡터) 태그를 피하고 다른 사람들이 이미 사용하고 있는 태그를 쓰는 편이 좋습니다. (플러그인 웹사이트를 참조하세요.)
- 기본 아이콘을 내버려두지 말고 제대로 된 아이콘을 추가해주십시오. 사용할 모범적인 스타일을 알고 싶다면 QGIS 인터페이스를 참조하세요.

16.4.2 코드와 도움말

- 저장소에 생성된 파일 (`ui_*.py`, `resources_rc.py`, 생성된 도움말 파일, …) 및 쓸모없는 것 (예: `.gitignore`) 들을 포함시키지 말아주십시오.
- 알맞은 메뉴 (Vector, Raster, Web, Database) 에 플러그인을 추가하십시오.
- 적절한 경우 (플러그인이 분석을 수행하는 경우) 플러그인을 공간 처리 프레임워크의 하위 플러그인으로 추가하는 것을 고려해주십시오. 그러면 사용자가 플러그인을 배치 (batch) 모드로 실행하거나 좀 더 복잡한 워크플로에 통합시킬 수 있고, 여러분이 인터페이스를 설계할 필요가 없어집니다.
- 적어도 최소한의 문서를 그리고 테스트 및 사용자 이해에 도움이 된다면 샘플 데이터를 포함시켜주십시오.

16.4.3 공식 파이썬 플러그인 저장소

공식 파이썬 플러그인 저장소는 https://plugins.qgis.org/ 입니다.

공식 저장소를 사용하려면 OSGEO 웹 포털 에서 OSGEO ID 를 만들어야만 합니다.

여러분의 플러그인을 업로드하고 나면 스태프 멤버가 이를 승인하고 여러분에게 알려줄 것입니다.

할 일:
관리 문서 (governance document) 를 가리키는 링크 삽입하기

권한

공식 플러그인 저장소에는 다음 규칙들이 구현되어 있습니다:

- 등록된 사용자 모두가 새 플러그인을 추가할 수 있습니다.
- 스태프 사용자는 모든 플러그인 버전들을 승인하거나 승인 취소할 수 있습니다.
- `plugins.can_approve` 특수 권한을 가진 사용자가 업로드한 버전은 자동으로 승인됩니다.
- `plugins.can_approve` 특수 권한을 가진 사용자는 플러그인 소유자 *(owner)* 목록에 들어가 있는 한 다른 사람이 업로드한 버전을 승인할 수 있습니다.

- 스태프 멤버와 해당 플러그인 소유자 만이 플러그인을 삭제하고 편집할 수 있습니다.
- `plugins.can_approve` 특수 권한이 없는 사용자가 새 버전을 업로드한 경우, 해당 플러그인 버전은 자동으로 승인 취소됩니다.

신뢰 관리

스태프 멤버들은 프론트엔드 응용 프로그램을 통해 선택된 플러그인 개발자들에게 `plugins.can_approve` 권한을 설정해서 신뢰 *(trust)* 를 부여할 수 있습니다.

플러그인 상세 정보 뷰는 플러그인 개발자 또는 플러그인 소유자 에게 신뢰를 부여할 수 있는 직접 링크를 제공합니다.

무결성 검증

플러그인이 업로드되면 압축 패키지에서 플러그인의 메타데이터를 자동으로 가져와서 무결성을 검증합니다.

공식 저장소에 플러그인을 업로드하려 할 때 알고 있어야 할 무결성 검증 규칙 몇 개를 소개합니다:

1. 여러분의 플러그인을 담고 있는 주 폴더의 이름은 오직 ASCII 문자 (A-Z 및 a-z), 숫자, 그리고 밑줄 (_) 과 빼기 (-) 기호로만 이루어져야 하며, 숫자로 시작되어서는 안 됩니다.
2. `metadata.txt` 파일은 필수입니다.
3. 메타데이터 표 목록에 있는 필수 메타데이터가 전부 있어야만 합니다.
4. `version` 메타데이터 필드가 유일해야만 합니다.
5. 사용 허가서 파일을 반드시 포함시켜야만 합니다. 확장자 없이 (예를 들어 `LICENSE.txt` 가 아닌) `LICENSE` 라는 이름으로 저장하십시오.

플러그인 구조

무결성 검증 규칙에 따라 플러그인의 압축 (`.zip`) 패키지는 기능하는 플러그인으로써 승인받기 위한 특수 구조를 가져야만 합니다. 사용자의 플러그인 폴더 안에 플러그인을 압축 해제할 것이기 때문에, 다른 플러그인과 섞이지 않도록 `.zip` 파일 안에 자체 디렉터리를 가지고 있어야만 합니다. 필수 파일은 `metadata.txt`, `__init__.py` 그리고 `LICENSE` 입니다. 하지만 `README` 파일은 물론 플러그인을 표현할 아이콘도 있는 편이 좋습니다. 다음은 `plugin.zip` 파일이 어떻게 보이면 되는지에 대한 예시입니다.

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

파이썬 프로그래밍 언어로 플러그인을 생성할 수 있습니다. C++ 로 작성된 전통적인 플러그인에 비해 더 쉽게 플러그인을 작성하고, 이해하고, 유지하고 배포할 수 있습니다. 이는 파이썬 언어의 동적인 특성 덕분입니다.

파이썬 플러그인 목록은 QGIS 플러그인 관리자에서 C++ 플러그인과 함께 볼 수 있습니다. `~/(UserProfile)/python/plugins` 및 다음 경로에서 플러그인을 검색합니다:

- 유닉스/맥: `(qgis_prefix)/share/qgis/python/plugins`
- 윈도우: `(qgis_prefix)/python/plugins`

~ 및 `(UserProfile)` 이 어떻게 정의되는지 알고 싶다면 core_and_external_plugins 을 참조하세요.

참고

기존 디렉터리 경로에 `QGIS_PLUGINPATH` 를 설정하면, 플러그인 검색 경로 목록에 이 경로를 추가할 수 있습니다.

CHAPTER 17

공간 처리 플러그인 작성하기

여러분이 개발하려는 플러그인의 유형에 따라, 플러그인의 기능을 공간 처리 알고리즘으로 (또는 공간 처리 알고리즘 집합으로) 추가하는 편이 더 나을 수도 있습니다. QGIS 안에 더 잘 통합시킬 수 있고, (모델 작성자 또는 배치 (batch) 공간 처리 인터페이스 같은 공간 처리 프레임워크 구성 요소에서 실행할 수 있기 때문에) 추가 기능을 제공할 수 있으며, (공간 처리 프레임워크가 작업의 상당 부분을 처리해줄 것이기 때문에) 개발 시간도 절약될 것입니다.

이런 알고리즘을 배포하려면, 공간 처리 툴박스에 알고리즘을 추가하는 새 플러그인을 생성해야 합니다. 이 플러그인은 알고리즘 제공자를 담고 있어야 하는데, 플러그인이 인스턴스화될 때 이 알고리즘 제공자를 등록해야 합니다.

17.1 처음부터 생성하기

알고리즘 제공자를 담고 있는 플러그인을 처음부터 생성하려면, 플러그인 작성자 (Plugin Builder) 를 사용해서 다음 단계들을 따라가면 됩니다:

1. **플러그인 작성자** 플러그인을 설치하십시오.
2. 플러그인 작성자를 사용해서 새 플러그인을 생성하십시오. 플러그인 작성자가 사용할 템플릿을 고르라고 할 때"공간 처리 제공자 (Processing provider)"를 선택하십시오.
3. 생성된 플러그인은 단일 알고리즘을 가진 제공자를 담고 있습니다. 제공자 파일과 알고리즘 파일 둘 다 충분한 주석들과 제공자를 수정하고 추가적인 알고리즘을 추가하는 방법에 대한 정보를 담고 있습니다. 자세한 정보를 알고 싶다면 이 파일들을 참조하세요.

17.2 플러그인을 업데이트하기

공간 처리 프레임워크에 여러분의 기존 플러그인을 추가하고 싶은 경우, 몇몇 코드를 추가해야 합니다.

1. `metadata.txt` 파일에 다음 변수를 추가해야 합니다:

```
hasProcessingProvider=yes
```

2. `initGui` 메소드를 사용해서 플러그인을 초기 설정하는 파이썬 파일에서 몇 줄을 다음과 같이 조정해줘야 합니다:

```
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` 폴더를 생성하고 다음 파일 3 개를 생성할 수 있습니다:
   - 아무 내용도 없는 `__init__.py` 파일—이 파일은 무결한 파이썬 패키지를 만들기 위해 필요합니다.
   - `provider.py` 파일—이 파일이 공간 처리 제공자를 생성하고 여러분의 알고리즘을 알릴 것입니다.

```
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.
        """
        return 'yourplugin'

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    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` 파일—이 파일은 예시 알고리즘 파일을 담습니다. 이 파일에 스크립트 템플릿 파일 의 내용을 복사/붙여넣기한 다음 여러분의 필요에 따라 업데이트하십시오.

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
```

4. 이제 QGIS 에 플러그인을 다시 불러오면 공간 처리 툴박스와 모델 작성자에서 여러분의 예시 스크립트를 볼 수 있을 것입니다.

17.3 Implementing custom Processing algorithms

17.3.1 Creating a custom algorithm

Here's a simple example of a custom buffer algorithm:

```
from qgis.core import (
    QgsProcessingAlgorithm,
    QgsProcessingParameterFeatureSource,
    QgsProcessingParameterNumber,
    QgsProcessingParameterFeatureSink,
    QgsFeatureSink,
)

class BufferAlgorithm(QgsProcessingAlgorithm):

    INPUT = 'INPUT'
    DISTANCE = 'DISTANCE'
    OUTPUT = 'OUTPUT'

    def initAlgorithm(self, config=None):
        self.addParameter(QgsProcessingParameterFeatureSource(self.INPUT, 'Input layer
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
↪'))
        self.addParameter(QgsProcessingParameterNumber(self.DISTANCE, 'Buffer distance
↪', defaultValue=100.0))
        self.addParameter(QgsProcessingParameterFeatureSink(self.OUTPUT, 'Output layer
↪'))

    def processAlgorithm(self, parameters, context, feedback):
        source = self.parameterAsSource(parameters, self.INPUT, context)
        distance = self.parameterAsDouble(parameters, self.DISTANCE, context)
        (sink, dest_id) = self.parameterAsSink(parameters, self.OUTPUT, context,
                                               source.fields(), source.wkbType(),␣
↪source.sourceCrs())

        for f in source.getFeatures():
            f.setGeometry(f.geometry().buffer(distance, 5))
            sink.addFeature(f, QgsFeatureSink.FastInsert)

        return {self.OUTPUT: dest_id}

    def name(self):
        return 'buffer'

    def displayName(self):
        return 'Buffer Features'

    def group(self):
        return 'Examples'

    def groupId(self):
        return 'examples'

    def createInstance(self):
        return BufferAlgorithm()
```

17.3.2 Customizing the algorithm dialog

Custom dialogs are especially useful when working with nested or dynamic inputs, when parameters depend on external data sources such as APIs (e.g. dynamically populated dropdowns), or when you need advanced validation and custom layout behavior that isn't supported by the default Processing dialog. To override the default UI (e.g. for complex parameter types or dynamic logic), subclass `QgsProcessingAlgorithmDialogBase`. To render your custom UI in the standard Processing dialog window, you must call `self.setMainWidget(panel)`, where `panel` is a `QgsPanelWidget` containing your custom layout. This ensures your interface is correctly displayed and interacts properly with the Processing framework.

Here is an example that integrates signal management using QTimer for debounced input:

```
from qgis.PyQt.QtCore import Qt, QT_VERSION_STR, QTimer
from qgis.core import (
    QgsProcessingAlgorithm,
    QgsProcessingContext,
    QgsProcessingFeedback,
    Qgis,
)
from qgis.PyQt.QtWidgets import QWidget, QVBoxLayout, QLineEdit
from qgis import gui, processing
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
from datetime import datetime
from typing import Dict, Optional
from osgeo import gdal

class CustomAlgorithmDialog(gui.QgsProcessingAlgorithmDialogBase):
    def __init__(
        self,
        algorithm: QgsProcessingAlgorithm,
        parent: Optional[QWidget] = None,
        title: Optional[str] = None,
    ):
        super().__init__(
            parent,
            flags=Qt.WindowFlags(),
            mode=gui.QgsProcessingAlgorithmDialogBase.DialogMode.Single,
        )
        self.context = QgsProcessingContext()
        self.setAlgorithm(algorithm)
        self.setModal(True)
        self.setWindowTitle(title or algorithm.displayName())

        self.panel = gui.QgsPanelWidget()
        layout = self.buildDialog()
        self.panel.setLayout(layout)
        self.setMainWidget(self.panel)

        self.cancelButton().clicked.connect(self.reject)

    def buildDialog(self) -> QVBoxLayout:
        layout = QVBoxLayout()

        self.input = QLineEdit()

        # Set up a debounced signal using QTimer
        self._update_timer = QTimer(self, singleShot=True)
        self._update_timer.timeout.connect(self._on_collection_id_ready)
        self.input.textChanged.connect(self._on_collection_id_changed)

        layout.addWidget(self.input)

        return layout

    def _on_collection_id_changed(self):
        self._update_timer.start(500)  # Debounce input

    def _on_collection_id_ready(self):
        self.pushInfo("Fetching metadata for collection ID…")

    def getParameters(self) -> Dict:
        try:
            return {'DISTANCE': float(self.input.text())}
        except ValueError:
            raise ValueError("Invalid buffer distance")

    def processingContext(self):
        return self.context
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

    def createFeedback(self):
        return QgsProcessingFeedback()

    def runAlgorithm(self):
        context = self.processingContext()
        feedback = self.createFeedback()
        params = self.getParameters()

        self.pushDebugInfo(f"QGIS version: {Qgis.QGIS_VERSION}")
        self.pushDebugInfo(f"QGIS code revision: {Qgis.QGIS_DEV_VERSION}")
        self.pushDebugInfo(f"Qt version: {QT_VERSION_STR}")
        self.pushDebugInfo(f"GDAL version: {gdal.VersionInfo('--version')}")
        self.pushCommandInfo(f"Algorithm started at: {datetime.now().isoformat(timespec='seconds')}")
        self.pushCommandInfo(f"Algorithm '{self.algorithm().displayName()}' starting...")
        self.pushCommandInfo("Input parameters:")
        for k, v in params.items():
            self.pushCommandInfo(f"  {k}: {v}")

        results = processing.run(self.algorithm(), params, context=context, feedback=feedback)
        self.setResults(results)
        self.showLog()
```

To launch the custom dialog for a given algorithm, simply instantiate `CustomAlgorithmDialog` with your algorithm instance and call `exec()`:

```
dlg = CustomAlgorithmDialog(BufferAlgorithm())
dlg.exec()
```

17.3.3 Managing Qt Signals

When building reactive dialogs, manage signal connections carefully. The above pattern uses a QTimer to debounce input from the text field, preventing rapid repeated calls. This is especially useful when fetching metadata or updating UI elements based on user input. Always connect signals once (typically in `__init__`) and use `singleShot=True` to ensure the slot is triggered only once after a delay.

CHAPTER 18

플러그인 레이어 사용하기

힌트

PyQGIS 콘솔을 사용하지 않는 경우 이 페이지에 있는 코드 조각들을 다음과 같이 가져와야 합니다:

```
from qgis.core import (
    QgsPluginLayer,
    QgsPluginLayerType,
    QgsMapLayerRenderer,
    QgsApplication,
    QgsProject,
)

from qgis.PyQt.QtGui import QImage
```

여러분의 플러그인이 맵 레이어를 렌더링하기 위해 자신만의 메소드를 이용하는 경우, 이를 구현하는 데 `QgsPluginLayer` 클래스에 기반한 여러분만의 레이어 유형을 작성하는 것이 가장 좋을 수도 있습니다.

18.1 QgsPluginLayer 하위 클래스 만들기

다음은 최소한의 `QgsPluginLayer` 클래스를 구현하는 예시입니다. 워터마크 예제 플러그인 의 원본 코드를 기반으로 하고 있습니다.

이 구현의 일부로써, 사용자 정의 렌더링 작업자가 캔버스에 실제로 그려지는 내용을 정의합니다.

```
class WatermarkLayerRenderer(QgsMapLayerRenderer):

    def __init__(self, layerId, rendererContext):
        super().__init__(layerId, rendererContext)

    def render(self):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        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

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
```

다른 모든 맵 레이어와 마찬가지로, 프로젝트 및 캔버스에 플러그인 레이어를 추가할 수 있습니다.

```
plugin_layer = WatermarkPluginLayer()
QgsProject.instance().addMapLayer(plugin_layer)
```

이런 레이어를 담고 있는 프로젝트를 불러오는 경우, 팩토리 클래스가 필요합니다:

```
class WatermarkPluginLayerType(QgsPluginLayerType):

    def __init__(self):
        super().__init__(WatermarkPluginLayer.LAYER_TYPE)

    def createLayer(self):
        return WatermarkPluginLayer()

    # You can also add GUI code for displaying custom information
    # in the layer properties
    def showLayerProperties(self, layer):
        pass


# Keep a reference to the instance in Python so it won't
# be garbage collected
plt =  WatermarkPluginLayerType()

assert QgsApplication.pluginLayerRegistry().addPluginLayerType(plt)
```

CHAPTER 19

네트워크 분석 라이브러리

힌트

PyQGIS 콘솔을 사용하지 않는 경우 이 페이지에 있는 코드 조각들을 다음과 같이 가져와야 합니다:

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

네트워크 분석 라이브러리는 다음에 사용할 수 있습니다:

- 지리 데이터 (폴리라인 벡터 레이어) 로부터 수학적인 그래프를 생성할 수 있습니다.
- 그래프 이론으로부터 기본 메소드들을 (현재로서는 데이크스트라 (Dijkstra) 알고리즘만) 구현할 수 있습니다.

RoadGraph 핵심 플러그인에서 기본 함수들을 내보내서 네트워크 분석 알고리즘을 생성했습니다. 이제 플러그인에서 또는 파이썬 콘솔에서 직접 이 라이브러리의 메소드들을 사용할 수 있습니다.

19.1 일반 정보

이 라이브러리의 전형적인 용도를 간단히 설명하면 다음과 같습니다:

1. 지리 데이터 (일반적으로 폴리라인 벡터 레이어) 로부터 그래프 생성
2. 그래프 분석 실행
3. 분석 결과 이용 (예를 들어 분석 결과의 시각화 등)

19.2 그래프 작성하기

가장 먼저 해야 할 일은 입력 데이터를 준비하는, 다시 말해 벡터 레이어를 그래프로 변환하는 것입니다. 이후의 모든 작업은 레이어가 아니라 이 그래프를 사용할 것입니다.

어떤 폴리라인 벡터 레이어라도 소스로 사용할 수 있습니다. 폴리라인의 노드 (node) 는 그래프의 꼭짓점 (vertex) 이 되고, 폴리라인의 선분 (segment) 은 그래프의 변 (edge) 이 됩니다. 노드 몇 개가 동일한 좌표에 있을 경우 그 노드들은 동일한 그래프 꼭짓점이 됩니다. 따라서 공통 노드를 가진 2 개의 선분은 서로 연결됩니다.

또, 그래프를 생성하는 도중에 입력 벡터 레이어에 추가적인 포인트를 몇 개라도"고정 (fix)"(다른 용어로는"결속 (tie)") 시킬 수 있습니다. 각 추가 포인트에 대응하는, 가장 가까운 그래프 꼭짓점 또는 가장 가까운 그래프 변을 찾을 것입니다. 후자의 경우 변을 분할해서 새 꼭짓점을 추가합니다.

벡터 레이어의 속성과 변 길이를 그래프 변의 속성으로 쓸 수 있습니다.

빌더 프로그래밍 패턴을 사용해서 벡터 레이어를 그래프로 변환합니다. 소위 말하는 디렉터 (Director) 를 사용해서 그래프를 작성합니다. 현재 디렉터는 `QgsVectorLayerDirector` 클래스 하나뿐입니다. 디렉터는 라인 벡터 레이어로부터 그래프를 작성하는 데 쓰일 기본 설정을 설정하는데, 빌더가 이를 사용해서 그래프를 생성합니다. 현재 존재하는 빌더는 디렉터의 경우와 마찬가지로 `QgsGraph` 클래스 객체를 생성하는 `QgsGraphBuilder` 클래스 하나뿐입니다. 여러분은 BGL 또는 NetworkX 같은 라이브러리와 호환되는 그래프를 작성하는 사용자 정의 빌더를 구현하기를 원할 수도 있습니다.

변 (edge) 속성을 계산하기 위해서는 전략 (strategy) 프로그래밍 패턴을 사용합니다. 현재로서는 (경로 (route) 의 길이를 연산에 넣는) `QgsNetworkDistanceStrategy` 전략과 (속도도 연산에 넣는) `QgsNetworkSpeedStrategy` 전략만 사용할 수 있습니다. 필요한 모든 파라미터를 사용하는 사용자 정의 전략을 구현할 수도 있습니다. 예를 들어 RoadGraph 플러그인은 속성으로부터 변 길이와 속도 값을 사용해서 이동 시간을 계산하는 전략을 사용합니다.

이제 실제로 해봅시다.

가장 먼저, 이 라이브러리를 사용하려면 분석 모듈을 가져와야 합니다:

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

다음은 디렉터를 생성하는 몇 가지 방법의 예시입니다:

```
# 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)
```

디렉터를 작성하려면, 그래프 구조 및 각 도로 선분 상에서 허용되는 (일방통행인지 양방향인지, 순방향인지 역방향인지) 움직임에 대한 정보의 소스로 사용할 벡터 레이어를 전달해야 합니다. 이 호출은 다음처럼 보일 것입니다:

```
director = QgsVectorLayerDirector(vectorLayer,
                                  directionFieldId,
                                  directDirectionValue,
                                  reverseDirectionValue,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                                    bothDirectionValue,
                                    defaultDirection)
```

각 파라미터가 의미하는 바는 다음과 같습니다:

- `vectorLayer` —그래프를 작성하기 위해 사용하는 벡터 레이어
- `directionFieldId` —도로 방향에 관한 정보가 저장된 속성 테이블 필드의 인덱스. 값이 `-1` 일 경우 방향 정보를 전혀 사용하지 않습니다. 정수형입니다.
- `directDirectionValue` —순방향 (라인의 첫 번째 포인트에서 마지막 포인트로 이동하는) 도로의 필드 값. 문자열입니다.
- `reverseDirectionValue` —역방향 (라인의 마지막 포인트에서 첫 번째 포인트로 이동하는) 도로의 필드 값. 문자열입니다.
- `bothDirectionValue` —양방향 (첫 번째 포인트에서 마지막으로도 마지막에서 첫 번째로도 이동할 수 있는) 도로의 필드 값. 문자열입니다.
- `defaultDirection` —기본 도로 방향. `directionFieldId` 필드가 설정되지 않은 도로 또는 앞에서 설명한 값 3 개 가운데 어떤 값과도 다른 값을 가진 도로에 이 값이 쓰일 것입니다. 다음 값들을 사용할 수 있습니다:
  - `QgsVectorLayerDirector.DirectionForward` —순방향 일방통행
  - `QgsVectorLayerDirector.DirectionBackward` —역방향 일방통행
  - `QgsVectorLayerDirector.DirectionBoth` —양방향

그 다음 변의 속성을 계산하기 위한 전략을 생성해야 합니다:

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

그리고 디렉터에 이 전략을 알려줍니다:

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

이제 그래프를 생성할 빌더를 사용할 수 있습니다. `QgsGraphBuilder` 클래스 작성자 (constructor) 는 인자 몇 개를 받습니다:

- `crs` —사용할 좌표계. 필수 인자입니다.
- `otfEnabled` —"실시간 (on-the-fly)"재투영을 사용할지 여부. 기본값은 `True` (OTF 사용) 입니다.
- `topologyTolerance` —위상 허용 오차. 기본값은 0 입니다.
- `ellipsoidID` —사용할 타원체. 기본값은"WGS84"입니다.

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

또 분석 작업에 사용할 포인트 몇 개를 다음과 같이 정의할 수 있습니다:

```
startPoint = QgsPointXY(1179720.1871, 5419067.3507)
endPoint = QgsPointXY(1180616.0205, 5419745.7839)
```

이제 모든 준비가 끝났기 때문에, 그래프를 작성하고 이 포인트들을 그래프에"결속 (tie)"시킬 수 있습니다:

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

그래프를 만드는 데 시간이 좀 걸릴 수도 있습니다. (레이어에 있는 피처의 개수 및 레이어 크기에 따라 다릅니다.) `tiedPoints` 는"결속"된 포인트들의 좌표 목록입니다. 빌더의 작업이 완료되면 분석에 사용할 수 있는 그래프를 얻게 됩니다:

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

다음 코드를 사용하면 포인트들의 꼭짓점 인덱스들을 얻을 수 있습니다:

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

19.3 그래프 분석

네트워크 분석은 다음 두 가지 질문에 대한 답을 찾는 데 사용됩니다. 어떤 꼭짓점들이 연결되어 있는가? 그리고 어떻게 최단 경로를 찾을 것인가? 네트워크 분석 라이브러리는 이 문제를 해결하기 위해 데이크스트라 알고리즘 (Dijkstra's algorithm) 을 제공합니다.

데이크스트라 알고리즘은 그래프의 한 꼭짓점에서 다른 모든 꼭짓점으로 가는 최단 경로와 최적화 파라미터의 값을 찾습니다. 그 결과는 최단 경로 트리로 나타낼 수 있습니다.

최단 경로 트리는 다음 속성들을 가진 방향성 가중치 그래프 (더 정확하게는 트리) 입니다:

- 들어오는 변이 없는 꼭짓점은 단 하나, 트리의 루트 (root) 뿐입니다.
- 다른 모든 꼭짓점은 들어오는 변을 딱 하나 가지고 있습니다.
- 꼭짓점 A 에서 꼭짓점 B 에 도달할 수 있다면, A 에서 B 로의 경로는 사용할 수 있는 단 하나의 경로이며 이 그래프 상에서 최적 (최단) 경로입니다.

최단 경로 트리를 생성하려면 `QgsGraphAnalyzer` 클래스의 `shortestTree()` 와 `dijkstra()` 메소드를 사용하십시오. `dijkstra()` 메소드를 사용할 것을 권장합니다. 더 빠르고 메모리를 더 효율적으로 사용하기 때문입니다.

여러분이 최단 경로 트리를 탐색하고 싶은 경우 `shortestTree()` 메소드가 유용합니다. 이 메소드는 언제나 새 그래프 객체 (`QgsGraph` 클래스) 를 생성하며 다음 변수 3 개를 받습니다:

- `source` —입력 그래프
- `startVertexIdx` —트리 상에 있는 포인트의 인덱스 (트리의 루트)
- `criterionNum` —사용할 변 속성의 번호 (0 에서 시작)

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

The `dijkstra()` method has the same arguments, but returns a tuple of arrays:

- In the first array, element *n* contains index of the incoming edge or -1 if there are no incoming edges.
- In the second array, element *n* contains the distance from the root of the tree to vertex *n* or DOUBLE_MAX if vertex *n* is unreachable from the root.

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

다음은 `shortestTree()` 메소드로 생성한 그래프를 사용해서 최단 경로 트리를 표시하는 매우 단순한 코드입니다. (*Layers* 패널에서 라인스트링 레이어를 선택한 다음 좌표를 여러분 자신의 좌표로 대체하십시오.)

> **경고**
>
> 이 코드를 예제로써만 사용하십시오. 이 코드는 `QgsRubberBand` 클래스 객체를 많이 생성하기 때문에 대용량 데이터셋의 경우 느려질 수도 있습니다.

```
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)
  rb.addPoint (tree.vertex(tree.edge(i).fromVertex()).point())
  rb.addPoint (tree.vertex(tree.edge(i).toVertex()).point())
  i = i + 1
```

동일한 목적이지만 `dijkstra()` 메소드를 사용하는 코드입니다:

```
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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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 최단 경로 찾기

두 포인트 사이의 최적 경로를 찾기 위해 다음 접근법을 사용합니다. 두 포인트 (시작 포인트 A 와 종단 포인트 B) 모두 작성 시 그래프에"결속"됩니다. 그 다음 `shortestTree()` 또는 `dijkstra()` 메소드를 사용해서 트리의 루트가 시작 포인트 A 인 최단 경로 트리를 생성합니다. 이 트리에서 종단 포인트 B 를 찾아, 트리 안에서 B 에서 A 로 가는 경로를 탐색합니다. 이 전체 알고리즘을 다음처럼 작성할 수 있습니다:

```
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
```

이 시점에서 이 경로를 지나가는 동안 거치게 될 꼭짓점들의 역순 목록의 형태로 경로를 얻게 됩니다. (꼭짓점들이 종단 포인트에서 시작 포인트 순서로 역순으로 나열됩니다.)

다음은 `shortestTree()` 메소드를 사용하는 QGIS 파이썬 콘솔의 예시 코드입니다 (TOC(Table of Contents) 에 라인스트링 레이어를 불러와 선택한 다음 코드에 있는 좌표를 여러분의 좌표로 대체해야 할 수도 있습니다):

```
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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

startPoint = QgsPointXY(1179661.925139,5419188.074362)
endPoint = QgsPointXY(1180942.970617,5420040.097560)

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

graph = builder.graph()
idxStart = graph.findVertex(tStart)

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

idxStart = tree.findVertex(tStart)
idxEnd = tree.findVertex(tStop)

if idxEnd == -1:
    raise Exception('No route!')

# Add last point
route = [tree.vertex(idxEnd).point()]

# Iterate the graph
while idxEnd != idxStart:
    edgeIds = tree.vertex(idxEnd).incomingEdges()
    if len(edgeIds) == 0:
        break
    edge = tree.edge(edgeIds[0])
    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)
```

다음은 동일한 예시이지만 `dijkstra()` 메소드를 사용하는 코드입니다:

```
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())

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
startPoint = QgsPointXY(1179661.925139,5419188.074362)
endPoint = QgsPointXY(1180942.970617,5420040.097560)

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

graph = builder.graph()
idxStart = graph.findVertex(tStart)
idxEnd = graph.findVertex(tStop)

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

if tree[idxEnd] == -1:
    raise Exception('No route!')

# Total cost
cost = costs[idxEnd]

# Add last point
route = [graph.vertex(idxEnd).point()]

# Iterate the graph
while idxEnd != idxStart:
    idxEnd = graph.edge(tree[idxEnd]).fromVertex()
    route.insert(0, graph.vertex(idxEnd).point())

# Display
rb = QgsRubberBand(iface.mapCanvas())
rb.setColor(Qt.red)

# This may require coordinate transformation if project's CRS
# is different than layer's CRS
for p in route:
    rb.addPoint(p)
```

19.3.2 도달 가능 영역

꼭짓점 A 의 도달 가능 영역 (area of availability) 이란 꼭짓점 A 에서 접근할 수 있고, 꼭짓점 A 에서 이 꼭짓점들까지의 경로 비용이 지정된 값을 초과하지 않는, 그래프 꼭짓점들의 부분집합을 말합니다.

다음 질문을 통해 이를 더 명확히 알 수 있습니다. "소방서가 있다. 소방차가 5 분/10 분/15 분 안에 도착할 수 있는 도시의 구역은 어디인가?"이 질문에 대한 답이 바로 소방서의 도달 가능 영역입니다.

도달 가능 영역을 찾으려면 `QgsGraphAnalyzer` 클래스의 `dijkstra()` 메소드를 사용하면 됩니다. 비용 배열의 요소들을 사전 정의된 값과 비교하는 것으로 충분합니다. cost[i] 가 사전 정의 값 이하인 경우, 꼭짓점 i 는 도달 가능 영역 안에 있는 것이고, 초과하는 경우 밖에 있는 것입니다.

도달 가능 영역의 경계를 구하는 일은 좀 더 어려운 문제입니다. 하단 경계는 도달 가능한 꼭짓점들의 집합이고, 상단 경계는 도달 불가능한 꼭짓점들의 집합입니다. 사실 단순합니다—변의 소스 꼭짓점에 접근할 수 있고 변의 대상 꼭짓점에는 접근할 수 없는 최단 경로 트리의 변들을 바탕으로 한 도달 가능 경계입니다.

다음은 그 예시입니다:

```
director = QgsVectorLayerDirector(vectorLayer, -1, '', '', '', QgsVectorLayerDirector.
↪DirectionBoth)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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)
    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))
```

CHAPTER 20

QGIS 서버와 파이썬

20.1 소개

QGIS 서버에 대해 더 알고 싶다면, QGIS-Server-manual 를 읽어보십시오.

QGIS 서버는 서로 다른 세 가지로 이루어져 있습니다:

1. QGIS 서버 라이브러리: OGC 웹 서비스를 생성하기 위한 API 를 제공하는 라이브러리입니다.
2. QGIS 서버 FCGI: 웹 서버와 함께 일련의 (WMS, WFS, WCS 등등의) OGC 서비스들 및 OGC API 들 (WFS3/OAPIF) 을 구현하는 FCGI 바이너리 응용 프로그램 `qgis_mapserv.fcgi` 파일
3. QGIS 개발 서버: 일련의 (WMS, WFS, WCS 등등의) OGC 서비스들 및 OGC API 들 (WFS3/OAPIF) 을 구현하는 개발 서버 바이너리 응용 프로그램 `qgis_mapserver` 파일

PyQGIS 쿡북의 이 장에서는 첫 번째 항목에 집중할 것입니다. QGIS 서버 API 의 사용 사례를 들어 파이썬을 사용해서 어떻게 서버의 습성을 확장하거나 향상시키거나 사용자 정의할 수 있는지를, 또는 QGIS 서버 API 를 사용해서 어떻게 또다른 응용 프로그램 안에 QGIS 서버를 내장시킬 수 있는지를 보여줄 것입니다.

QGIS 서버의 습성을 변경하거나, 새로운 사용자 정의 서비스 또는 API 를 제공하기 위해 QGIS 서버의 케이퍼빌리티를 확장할 수 있는 서로 다른 방법들이 몇 가지 있습니다. 다음은 여러분이 마주할 수도 있는 주요 시나리오들입니다:

- EMBEDDING → 또다른 파이썬 응용 프로그램에서 QGIS 서버 API 사용하기
- STANDALONE → QGIS 서버를 독립형 WSGI/HTTP 서비스로 실행하기
- FILTERS → QGIS 서버를 필터 플러그인을 사용해서 향상시키기/사용자 정의하기
- SERVICES → 새 *SERVICE* 를 추가하기
- OGC APIs → 새 *OGC API* 를 추가하기

내장형 및 독립형 응용 프로그램들은 또다른 파이썬 스크립트 또는 응용 프로그램으로부터 QGIS 서버 파이썬 API 를 사용해야 합니다. 나머지 옵션들은 표준 QGIS 서버 바이너리 응용 프로그램 (FCGI 또는 개발 서버) 에 사용자 정의 기능들을 추가하고 싶은 경우 더 어울립니다. 이런 경우 서버 응용 프로그램 용 파이썬 플러그인을 작성한 다음 사용자 정의 필터, 서비스, 또는 API 를 등록해야 할 것입니다.

20.2 서버 API 기본 사항

전형적인 QGIS 서버 응용 프로그램과 관련된 기본 클래스는 다음과 같습니다:

- `QgsServer`: 서버 인스턴스 (일반적으로 응용 프로그램 전체 생애 주기에 대해 단일 인스턴스)
- `QgsServerRequest`: 요청 객체 (일반적으로 각 요청마다 다시 생성)
- `QgsServer.handleRequest(request, response)`: 이 메소드가 요청을 처리하고 응답을 채웁니다.

QGIS 서버 FCGI 또는 개발 서버 워크플로는 다음과 같이 요약할 수 있습니다:

```
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)` 메소드 안에서 필터 플러그인 콜백을 호출하고 `QgsServerInterface` 클래스를 통해 플러그인이 `QgsServerRequest` 및 `QgsServerResponse` 클래스들을 사용할 수 있게 됩니다.

> **경고**
>
> QGIS 서버 클래스들은 스레드 안전 (thread safety) 하지 않기 때문에, QGIS 서버 API 를 기반으로 확장 가능한 (scalable) 응용 프로그램을 작성할 때 항상 다중 처리 (multiprocessing) 모델 또는 컨테이너를 사용해야 합니다.

20.3 독립형 또는 내장형

독립형 또는 내장형 서버 응용 프로그램의 경우, 앞에서 언급했던 서버 클래스들을 클라이언트와의 모든 HTTP 프로토콜 상호 작용들을 관리하는 웹 서버 구현으로 래핑 (wrapping) 해서 직접 사용해야 합니다.

QGIS 서버 API 사용 사례의 (HTTP 부분을 뺀) 최소한의 예시는 다음과 같습니다:

```
from qgis.core import QgsApplication
from qgis.server import *
app = QgsApplication([], False)

# Create the server instance, it may be a single one that
# is reused on multiple requests
server = QgsServer()

# Create the request by specifying the full URL and an optional body
# (for example for POST requests)
request = QgsBufferServerRequest(
    'http://localhost:8081/?MAP=/qgis-server/projects/helloworld.qgs' +
    '&SERVICE=WMS&REQUEST=GetCapabilities')

# Create a response objects
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
response = QgsBufferServerResponse()

# Handle the request
server.handleRequest(request, response)

print(response.headers())
print(response.body().data().decode('utf8'))

app.exitQgis()
```

다음은 QGIS 소스 코드 저장소 상에서 CI(continuous integration) 를 테스트하기 위해 개발된 완전한 독립형 응용 프로그램의 예시로써, 서로 다른 플러그인 필터들과 인증 스키마들의 광범위한 집합을 보여줍니다 (테스트 목적만으로 개발되었기 때문에 생산 용은 아니지만, 학습 용으로도 흥미롭습니다): qgis_wrapped_server.py

20.4 서버 플러그인

QGIS 서버 응용 프로그램을 구동하고 필터, 서비스, 또는 API 를 등록하기 위해 사용할 수 있게 되면, 서버 파이썬 플러그인을 불러옵니다.

서버 플러그인의 구조는 대응하는 데스크탑 플러그인과 매우 유사합니다. 플러그인이 `QgsServerInterface` 객체를 사용할 수 있게 해주고, 플러그인이 대응하는 레지스트리에 하나 이상의 사용자 정의 필터, 서비스, 또는 API 를 서버 인터페이스에서 볼 수 있는 메소드들 가운데 하나를 사용해서 등록할 수 있습니다.

20.4.1 서버 필터 플러그인

필터는 세 가지 종류가 있으며, 다음 클래스 가운데 하나를 하위 클래스화하고 `QgsServerInterface` 클래스의 대응하는 메소드를 호출해서 인스턴스화할 수 있습니다:

필터 유형	기반 클래스	QgsServerInterface 등록
I/O	`QgsServerFilter`	`registerFilter()`
접근 제어	`QgsAccessControlFilter`	`registerAccessControl()`
캐시	`QgsServerCacheFilter`	`registerServerCache()`

I/O 필터

I/O 필터는 (WMS, WFS 등등의) 핵심 서비스들의 서버 입력 및 출력을 (요청 및 응답을) 조정할 수 있기 때문에 서비스 워크플로를 어떤 유형으로든 조작할 수 있게 해줍니다. 예를 들면 선택한 레이어에의 접근을 제한할 수도 있고, XML 응답에 XSL 스타일시트를 주입 (inject) 할 수도 있으며, 생성한 WMS 이미지에 워터마크를 추가할 수도 있습니다.

이 시점에서 서버 플러그인 API 문서 를 한번 훑어보는 편이 유용할 수도 있습니다.

각 필터는 다음 콜백 3 개 가운데 적어도 하나를 구현해야 합니다:

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

모든 필터는 요청/응답 객체 (`QgsRequestHandler`) 에 접근할 수 있고 이 객체의 모든 속성 (입력/출력) 을 조작할 수 있으며 (다음에서 보게 될 아주 특수한 방식이긴 하지만) 예외를 발생시킬 수 있습니다.

이 메소드들은 모두 호출을 후속 필터들까지 전파해야 하는지 여부를 나타내는 불 (boolean) 값을 반환합니다. 이 메소드들 가운데 하나가 `False` 를 반환하면 연쇄 전파가 중단되고, 그렇지 않으면 호출을 다음 필터로 전파할 것입니다.

다음은 서버가 전형적인 요청을 어떻게 처리하는지 그리고 언제 필터의 콜백을 호출하는지를 보여주는 의사 (pseudo) 코드입니다:

```
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
```

다음 문단들에서 사용할 수 있는 콜백들을 자세히 설명합니다.

onRequestReady

요청이 준비됐을 때 이 콜백을 호출합니다. 들어오는 URL 과 데이터를 파싱한 다음 핵심 서비스 (WMS, WFS 등등) 스위치를 입력하기 전, 이 때가 여러분이 입력을 조작하고 다음과 같은 액션들을 수행할 수 있는 시점입니다:

- 인증/승인
- 리다이렉트
- 특정 파라미터 (예를 들면 typename) 추가/제거
- 예외 발생

SERVICE 파라미터를 변경, 핵심 서비스를 완전히 우회 (bypass) 해서 핵심 서비스조차도 완전히 대체할 수 있습니다. (물론 큰 의미가 있는 방법은 아닙니다.)

onSendResponse

어떤 부분적인 출력이라도 응답 버퍼에서 (예를 들어 FCGI 서버를 사용하는 경우 **FCGI** `stdout` 으로) 플러시 (flush) 될 때마다 이 콜백을 호출하고, 그 다음 클라이언트로 전달합니다. 대용량 콘텐츠가 스트리밍 될 때 (예: WFS GetFeature) 이런 일이 발생합니다. 이 경우 `onSendResponse()` 를 여러 번 호출할 수도 있습니다.

응답을 스트리밍하지 않는 경우 `onSendResponse()` 메소드를 전혀 호출하지 않을 것이란 점을 기억하십시오.

모든 경우에, `onResponseComplete()` 메소드를 호출한 다음 클라이언트에 마지막 (또는 유일한) 덩어리 (chunk) 를 전달할 것입니다.

`False` 를 반환하면 클라이언트에 데이터를 플러시하지 않을 것입니다. 플러그인이 응답에서 나오는 모든 덩어리들을 모은 다음 검사하거나 또는 `onResponseComplete()` 에서 응답을 변경하고자 하는 경우 유용합니다.

onResponseComplete

핵심 서비스가 (적중한 경우) 처리 과정을 완료하고 클라이언트에 응답을 전달할 준비가 됐을 때 이 콜백을 한 번 호출합니다. 앞에서 설명한 대로, 클라이언트에 마지막 (또는 유일한) 데이터 덩어리를 전달하기 전에 이 메소드를 호출할 것입니다. 스트리밍 서비스의 경우, `onSendResponse()` 메소드를 여러 번 호출할 수도 있습니다.

`onResponseComplete()` 가 새로운 서비스를 (WPS 또는 사용자 정의 서비스를) 구현하고 핵심 서비스에서 나오는 출력을 직접 조작할 수 있는 (예를 들면 WMS 이미지 위에 워터마크를 추가할 수 있는) 이상적인 위치입니다.

`False` 를 반환하면 다음 플러그인들이 `onResponseComplete()` 메소드를 실행하지 않을 것이지만, 어떤 경우에라도 클라이언트에 응답을 전달하지 않을 것이라는 사실을 기억하십시오.

플러그인에서 예외를 발생시키기

이 주제에 대해서는 아직 좀 더 작업을 해야 합니다. 현재 구현된 바로는 `QgsMapServiceException` 클래스의 인스턴스에 `QgsRequestHandler` 속성을 설정해서 처리된 예외와 처리되지 않은 예외를 구별할 수 있습니다. 이런 방식은 주 C++ 코드가 처리된 파이썬 예외를 캐치하고 처리되지 않은 예외는 무시 (또는 더 나은 방법으로는 로그를 작성) 할 수 있게 합니다.

이 접근법은 기본적으로 작동하지만 그다지"파이썬스럽지"않습니다. 더 나은 접근법은 파이썬 코드로부터 예외를 발생시킨 다음 처리를 위해 C++ 루프로 버블링 (bubbling) 시키는 것일 겁니다.

서버 플러그인 작성하기

서버 플러그인은파이썬 플러그인 개발하기 에서 설명한 대로 추가적인 (또는 대체하는) 인터페이스만 제공하는 표준 QGIS 파이썬 플러그인입니다. 일반적인 QGIS 데스크탑 플러그인은 `QgisInterface` 인스턴스를 통해 QGIS 응용 프로그램에 접근할 수 있지만, 서버 플러그인은 QGIS 서버 응용 프로그램 맥락 안에서 실행될 때에만 `QgsServerInterface` 클래스에 접근할 수 있습니다.

QGIS 서버가 플러그인이 서버 인터페이스를 가지고 있다는 사실을 알게 하려면, (`metadata.txt` 파일에) 특별한 메타데이터 항목이 필요합니다:

```
server=True
```

> **중요**
>
> QGIS 서버는 `server=True` 메타데이터를 가진 플러그인들만 불러오고 실행할 것입니다.

여기에서 설명하는 (더 많은 플러그인을 가진) qgis3-server-vagrant 예제 플러그인은 깃허브에서 사용할 수 있으며, 몇몇 서버 플러그인은 공식 QGIS 플러그인 저장소 에도 공개되어 있습니다.

플러그인 파일들

다음은 예제 서버 플러그인의 디렉터리 구조입니다.

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

__init__.py

파이썬의 가져오기 시스템이 이 파일을 필요로 합니다. 또 QGIS 서버도 이 파일이 `serverClassFactory()` 함수를 담고 있을 것을 요구합니다. 이 함수는 서버 구동 시 QGIS 서버에 플러그인을 불러왔을 때 호출되어, `QgisServerInterface` 클래스의 인스턴스를 가리키는 참조를 받아서 플러그인 클래스의 인스턴스를 반환해야만 합니다. `__init__.py` 의 내용은 다음과 같이 보여야 합니다:

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

HelloServer.py

이 파일에서 마법이 벌어지는데, 이 마법은 다음과 같이 보일 것입니다: (`HelloServer.py` 파일의 예시)

서버 플러그인은 일반적으로 `QgsServerFilter` 클래스의 인스턴스에 패키징된 하나 이상의 콜백으로 이루어져 있습니다.

각각의 `QgsServerFilter` 클래스는 다음과 같은 하나 이상의 콜백을 구현합니다:

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

다음은 **SERVICE** 파라미터가"HELLO"와 동등한 경우 *HelloServer!* 를 출력하는 최소한의 필터를 구현하는 예시입니다:

```
class HelloFilter(QgsServerFilter):

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

    def onRequestReady(self) -> bool:
        QgsMessageLog.logMessage("HelloFilter.onRequestReady")
        return True

    def onSendResponse(self) -> bool:
        QgsMessageLog.logMessage("HelloFilter.onSendResponse")
        return True

    def onResponseComplete(self) -> bool:
        QgsMessageLog.logMessage("HelloFilter.onResponseComplete")
        request = self.serverInterface().requestHandler()
        params = request.parameterMap()
        if params.get('SERVICE', '').upper() == 'HELLO':
            request.clear()
            request.setResponseHeader('Content-type', 'text/plain')
            # Note that the content is of type "bytes"
            request.appendBody(b'HelloServer!')
        return True
```

이 필터들은 다음 예시에서처럼 반드시 **serverIface** 에 등록되어 있어야만 합니다:

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

registerFilter() 메소드의 두 번째 파라미터는 같은 이름을 가진 콜백들에 대한 순서를 정의하는 우선 순위를 설정합니다 (우선 순위 값이 낮을수록 먼저 호출됩니다):

플러그인은 이 3 개의 콜백을 사용해서 서버의 입력 그리고/또는 출력을 서로 다른 많은 방식으로 조작할 수 있습니다. 모든 경우에서, 플러그인 인스턴스는 QgsServerInterface 클래스를 통해 QgsRequestHandler 클래스에 접근합니다. QgsRequestHandler 클래스는 (requestReady() 함수를 사용해서) 서버의 핵심 처리 과정에 들어가기 전에 또는 (sendResponse() 함수를 사용해서) 핵심 서비스가 요청을 처리한 후에 입력 파라미터들을 변경하는 데 쓰일 수 있는 메소드들을 많이 가지고 있습니다.

다음은 흔한 사용 사례 몇 개를 설명하는 예시들입니다:

입력을 수정하기

다음 플러그인은 쿼리 문자열에서 나오는 입력 파라미터들을 변경하는 테스트 예제로, 이 예시에서는 (이미 파싱된) parameterMap 에 새 파라미터를 주입합니다. 그러면 (WMS 등등의) 핵심 서비스가 이 파라미터를 인식해서 핵심 서비스 처리 과정 마지막에 해당 파라미터가 그대로 있는지 확인합니다:

```
class ParamsFilter(QgsServerFilter):

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

    def onRequestReady(self) -> bool:
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        request.setParameter('TEST_NEW_PARAM', 'ParamsFilter')
        return True

    def onResponseComplete(self) -> bool:
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        if params.get('TEST_NEW_PARAM') == 'ParamsFilter':
            QgsMessageLog.logMessage("SUCCESS - ParamsFilter.onResponseComplete")
        else:
            QgsMessageLog.logMessage("FAIL    - ParamsFilter.onResponseComplete")
        return True
```

다음은 로그 파일에 보이는 내용을 추출한 것입니다:

```
 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
```

강조된 줄에 있는"SUCCESS"문자열이 플러그인이 테스트를 통과했다는 사실을 나타냅니다.

동일한 기술을 사용해서 핵심 서비스 대신 사용자 정의 서비스를 사용할 수 있습니다. 예를 들어 **SERVICE** 파라미터를 다른 무언가로 변경하는 것만으로도 **WFS SERVICE** 요청 또는 다른 모든 핵심 요청을 뛰어넘을 수 있으며, 그러면 핵심 서비스를 뛰어넘을 것입니다. 그 다음 출력에 사용자 정의 결과를 주입해서 클라이언트에 전달할 수 있습니다. (이에 대해서는 다음 부분에서 설명할 것입니다.)

> **팁**
>
> 여러분이 정말로 사용자 정의 서비스를 구현하고자 하는 경우, `QgsService` 클래스의 하위 클래스를 생성한 다음 `registerService(service)` 메소드를 호출해서 `registerFilter()` 메소드 상에서 사용자 정의 서비스를 등록하는 방식을 권장합니다.

출력을 수정 또는 대체하기

다음 워터마크 필터 예시는 WMS 출력을 WMS 핵심 서비스가 생성한 WMS 이미지 위에 워터마크 이미지를 추가해서 만든 새 이미지로 대체하는 방법을 보여줍니다:

```
from qgis.server import *
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *

class WatermarkFilter(QgsServerFilter):

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

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

이 예시에서는 **SERVICE** 파라미터 값을 확인해서 들어오는 요청이 **WMS GETMAP** 이며 이전에 실행된 플러그인이 또는 핵심 서비스가 (이 경우 WMS 가) 어떤 예외도 설정하지 않은 경우, 출력 버퍼에서 WMS 가 생성한 이미지를 검색해서 워터마크 이미지를 추가합니다. 마지막 단계는 출력 버퍼를 지운 다음 새로 생성한 이미지로 대체하는 것입니다. 실제 상황에서는 PNG 또는 JPG 만 지원하는 것이 아니라 요청된 이미지 유형도 확인해야 한다는 사실을 기억하십시오.

접근 제어 필터

접근 제어 필터들은 개발자가 어떤 레이어, 필터, 그리고 속성에 접근할 수 있는지에 대해 세밀하게 제어할 수 있게 해줍니다. 접근 제어 필터에 다음 콜백들을 구현할 수 있습니다:

- `layerFilterExpression(layer)`
- `layerFilterSubsetString(layer)`
- `layerPermissions(layer)`
- `authorizedLayerAttributes(layer, attributes)`
- `allowToEdit(layer, feature)`
- `cacheKey()`

플러그인 파일들

다음은 예제 플러그인의 디렉터리 구조입니다.

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

__init__.py

파이썬의 가져오기 시스템이 이 파일을 필요로 합니다. 모든 QGIS 서버 플러그인과 마찬가지로, 이 파일이 `serverClassFactory()` 함수를 담고 있습니다. 이 함수는 서버 구동 시 QGIS 서버에 플러그인을 불러왔을 때 호출되어, `QgisServerInterface` 클래스의 인스턴스를 가리키는 참조를 받아서 플러그인 클래스의 인스턴스를 반환해야만 합니다. `__init__.py` 의 내용은 다음과 같이 보여야 합니다:

```
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):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        """ Return the layer rights """
        return super().layerPermissions(layer)

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

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

    def cacheKey(self):
        return super().cacheKey()

class AccessControlServer:

   def __init__(self, serverIface):
      """ Register AccessControlFilter """
      serverIface.registerAccessControl(AccessControlFilter(serverIface), 100)
```

이 예시는 모두에게 완전한 접근 권한을 부여합니다.

누가 로그인했는지 확인하는 것이 플러그인의 역할입니다.

이런 모든 메소드에는 제약 조건을 레이어 별로 사용자 정의할 수 있는 레이어 인자가 있습니다.

layerFilterExpression

결과를 제한하기 위한 표현식을 추가하는 데 쓰입니다.

예를 들면 `role` 속성이 `user` 인 경우로 피처를 제한할 수 있습니다.

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

layerFilterSubsetString

바로 앞의 함수와 동일하지만 (데이터베이스에서 실행되는) `SubsetString` 을 사용합니다.

예를 들면 `role` 속성이 `user` 인 경우로 피처를 제한할 수 있습니다.

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

layerPermissions

레이어에의 접근을 제한합니다.

다음 속성을 가진 `LayerPermissions()` 유형의 객체를 반환합니다:

- `canRead`: `GetCapabilities` 에서 피처를 볼 수 있고 읽기 접근 권한을 가집니다.
- `canInsert`: 새 피처를 삽입할 수 있습니다.
- `canUpdate`: 피처를 업데이트할 수 있습니다.
- `canDelete`: 피처를 삭제할 수 있습니다.

예를 들어 읽기 전용 접근에서 모든 권한을 제한하려면:

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

authorizedLayerAttributes

속성의 특정 하위 집합의 가시성을 제한하는 데 쓰입니다.

이 인자 속성은 현재 가시 속성 집합을 반환합니다.

예를 들어 `role` 속성을 숨기려면:

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

allowToEdit

피처들의 하위 집합에 대한 편집 작업을 제한하는 데 쓰입니다.

이 함수는 `WFS-Transaction` 프로토콜에 사용됩니다.

예를 들면 `role` 속성이 `user` 인 피처만 편집할 수 있게 하려면:

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

cacheKey

QGSI 서버가 케이퍼빌리티의 캐시를 유지·관리하는 경우 역할 (role) 별로 캐시하려면 이 메소드에서 역할을 반환하면 됩니다. 또는 `None` 을 반환해서 캐시를 완전히 비활성화시킬 수도 있습니다.

20.4.2 사용자 정의 서비스

QGIS 서버에서 WMS, WFS 및 WCS 같은 핵심 서비스들은 `QgsService` 클래스의 하위 클래스로 구현됩니다.

`SERVICE` 쿼리 문자열 파라미터가 서비스 이름과 일치할 때 실행될 새 서비스를 구현하려면, 여러분 고유의 `QgsService` 클래스를 구현한 다음 `registerService(service)` 메소드를 호출해서 `serviceRegistry()` 메소드 상에서 여러분의 서비스를 등록하면 됩니다.

다음은 `CUSTOM` 이라는 이름을 가진 사용자 정의 서비스의 예시입니다:

```
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 사용자 정의 API

QGIS 서버에서, OAPIF (다른 이름으로는 WFS3) 같은 OGC API 들은 `QgsServerOgcApi` 클래스의 인스턴스에 등록된 `QgsServerOgcApiHandler` 하위 클래스들의 집합으로 (또는 그 상위 클래스인 `QgsServerApi` 로) 구현됩니다.

URL 경로가 특정 URL 과 일치할 때 실행될 새 API 를 구현하려면, 여러분 고유의 `QgsServerOgcApiHandler` 클래스 인스턴스들을 구현한 다음 `QgsServerOgcApi` 클래스에 추가하고 `registerApi(api)` 메소드를 호출해서 `serviceRegistry()` 메소드 상에서 API 를 등록하면 됩니다.

다음은 URL 이 `/customapi` 를 담고 있는 경우 실행될 사용자 정의 API 의 예시입니다:

```
import json
import os

from qgis.PyQt.QtCore import QBuffer, QIODevice, QTextStream, QRegularExpression
from qgis.server import (
    QgsServiceRegistry,
    QgsService,
    QgsServerFilter,
    QgsServerOgcApi,
    QgsServerQueryStringParameter,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    QgsServerOgcApiHandler,
)

from qgis.core import (
    QgsMessageLog,
    QgsJsonExporter,
    QgsCircle,
    QgsFeature,
    QgsPoint,
    QgsGeometry,
)


class CustomApiHandler(QgsServerOgcApiHandler):

    def __init__(self):
        super(CustomApiHandler, self).__init__()
        self.setContentTypes([QgsServerOgcApi.HTML, QgsServerOgcApi.JSON])

    def path(self):
        return QRegularExpression("/customapi")

    def operationId(self):
        return "CustomApiXYCircle"

    def summary(self):
        return "Creates a circle around a point"

    def description(self):
        return "Creates a circle around a point"

    def linkTitle(self):
        return "Custom Api XY Circle"

    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,␣
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
↪QgsServerQueryStringParameter.Type.Double, 'X coordinate'),
                QgsServerQueryStringParameter(
                    'y', True, QgsServerQueryStringParameter.Type.Double, 'Y␣
↪coordinate'),
                QgsServerQueryStringParameter('r', True,␣
↪QgsServerQueryStringParameter.Type.Double, 'radius')]


class CustomApi():

    def __init__(self, serverIface):
        api = QgsServerOgcApi(serverIface, '/customapi',
                              'custom api', 'a custom api', '1.1')
        handler = CustomApiHandler()
        api.registerHandler(handler)
        serverIface.serviceRegistry().registerApi(api)
```

CHAPTER 21

PyQGIS 의 꼼수와 편법

힌트

PyQGIS 콘솔을 사용하지 않는 경우 이 페이지에 있는 코드 조각들을 다음과 같이 가져와야 합니다:

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

from qgis.core import (
    Qgis,
    QgsProject,
    QgsLayerTreeModel,
)

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

21.1 사용자 인터페이스

룩 앤드 필 바꾸기

```
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())
```

아이콘과 제목 바꾸기

```
from qgis.PyQt.QtGui import QIcon

icon = QIcon("/path/to/logo/file.png")
iface.mainWindow().setWindowIcon(icon)
iface.mainWindow().setWindowTitle("My QGIS")
```

21.2 설정

QgsSettings 목록 얻기

```
from qgis.core import QgsSettings

qs = QgsSettings()

for k in sorted(qs.allKeys()):
    print (k)
```

21.3 툴바

툴바 제거하기

```
toolbar = iface.helpToolBar()
parent = toolbar.parentWidget()
parent.removeToolBar(toolbar)

# and add again
parent.addToolBar(toolbar)
```

액션 툴바 제거하기

```
actions = iface.attributesToolBar().actions()
iface.attributesToolBar().clear()
iface.attributesToolBar().addAction(actions[4])
iface.attributesToolBar().addAction(actions[3])
```

21.4 메뉴 그룹

메뉴 제거하기

```
# for example Help Menu
menu = iface.helpMenu()
menubar = menu.parentWidget()
menubar.removeAction(menu.menuAction())

# and add again
menubar.addAction(menu.menuAction())
```

21.5 캔버스

캔버스에 접근하기

```
canvas = iface.mapCanvas()
```

캔버스 색상 바꾸기

```
from qgis.PyQt.QtCore import Qt

iface.mapCanvas().setCanvasColor(Qt.black)
iface.mapCanvas().refresh()
```

맵 업데이트 주기

```
from qgis.core import QgsSettings
# Set milliseconds (150 milliseconds)
QgsSettings().setValue("/qgis/map_update_interval", 150)
```

21.6 레이어

벡터 레이어 추가하기

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

활성 레이어 얻기

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

모든 레이어를 목록화하기

```
from qgis.core import QgsProject

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

레이어 이름 얻기

```
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']
```

다른 방법

```
layers_names = [layer.name() for layer in QgsProject.instance().mapLayers().values()]
print("layers TOC = {}".format(layers_names))
```

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

이름으로 레이어 찾기

```
from qgis.core import QgsProject

layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
print(layer.name())
```

```
layer name you like
```

활성 레이어 설정하기

```
from qgis.core import QgsProject

layer = QgsProject.instance().mapLayersByName("layer name you like")[0]
iface.setActiveLayer(layer)
```

레이어를 주기적으로 새로고침하기

```
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)
```

메소드 보이기

```
dir(layer)
```

피처 양식으로 새 피처 추가하기

```
from qgis.core import QgsFeature, QgsGeometry

feat = QgsFeature()
geom = QgsGeometry()
feat.setGeometry(geom)
feat.setFields(layer.fields())

iface.openFeatureForm(layer, feat, False)
```

피처 양식 없이 새 피처 추가하기

```
from qgis.core import QgsGeometry, QgsPointXY, QgsFeature

pr = layer.dataProvider()
feat = QgsFeature()
feat.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(10,10)))
pr.addFeatures([feat])
```

피처 얻기

```
for f in layer.getFeatures():
    print (f)
```

```
<qgis._core.QgsFeature object at 0x7f45cc64b678>
```

선택한 피처 얻기

```
for f in layer.selectedFeatures():
    print (f)
```

선택한 피처 ID 얻기

```
selected_ids = layer.selectedFeatureIds()
print(selected_ids)
```

선택한 피처 ID 로부터 메모리 레이어 생성하기

```
from qgis.core import QgsFeatureRequest

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

도형 얻기

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

```
10.000000, 10.000000
```

도형 이동시키기

```
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)>
```

좌표계 설정하기

```
from qgis.core import QgsProject, QgsCoordinateReferenceSystem

for layer in QgsProject.instance().mapLayers().values():
    layer.setCrs(QgsCoordinateReferenceSystem('EPSG:4326'))
```

좌표계 보기

```
from qgis.core import QgsProject

for layer in QgsProject.instance().mapLayers().values():
    crs = layer.crs().authid()
    layer.setName('{} ({})'.format(layer.name(), crs))
```

필드 열 숨기기

```
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 에서 나온 레이어

```
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 에서 벡터 레이어를 모두 불러오기

```
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)
```

타일 레이어 (XYZ 레이어) 불러오기

```
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')
```

레이어를 모두 제거하기

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

모두 제거하기

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

21.7 TOC(Table of Contents)

체크한 레이어에 접근하기

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

컨텍스트 메뉴 제거하기

```
ltv = iface.layerTreeView()
mp = ltv.menuProvider()
ltv.setMenuProvider(None)
# Restore
ltv.setMenuProvider(mp)
```

21.8 고급 TOC

루트 노드

```
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())
```

첫 번째 하위 노드에 접근하기

```
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
```

그룹과 노드 찾기

```
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
```

이름으로 그룹 찾기

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

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

ID 로 레이어 찾기

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

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

레이어 추가하기

```
from qgis.core import QgsVectorLayer, QgsProject

layer1 = QgsVectorLayer("Point?crs=EPSG:4326", "layer name you like 2", "memory")
QgsProject.instance().addMapLayer(layer1, False)
node_layer1 = root.addLayer(layer1)
# Remove it
QgsProject.instance().removeMapLayer(layer1)
```

그룹 추가하기

```
from qgis.core import QgsLayerTreeGroup

node_group2 = QgsLayerTreeGroup("Group 2")
root.addChildNode(node_group2)
QgsProject.instance().mapLayersByName("layer name you like")[0]
```

불러온 레이어 이동시키기

```
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)
```

불러온 레이어를 특정 그룹으로 이동시키기

```
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)
```

활성 레이어의 가시성을 켜고끄기

```
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)
```

그룹이 선택되었는지 여부

```
def isMyGroupSelected( groupName ):
    myGroup = QgsProject.instance().layerTreeRoot().findGroup( groupName )
    return myGroup in iface.layerTreeView().selectedNodes()

print(isMyGroupSelected( 'my group name' ))
```

```
False
```

노드 펼치기

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

숨긴 노드 꼼수

```
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())

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
index = model.node2index( node )
ltv.setRowHidden( index.row(), index.parent(), True )
node.setCustomProperty( 'nodeHidden', 'true')
ltv.setCurrentIndex(model.node2index(root))
```

노드 신호

```
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)
```

레이어 제거하기

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

그룹 제거하기

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

새 TOC 생성하기

```
root = QgsProject.instance().layerTreeRoot()
model = QgsLayerTreeModel(root)
view = QgsLayerTreeView()
view.setModel(model)
view.show()
```

노드 이동시키기

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

노드 이름 바꾸기

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

21.9 공간 처리 알고리즘

알고리즘 목록 얻기

```
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
```

알고리즘 도움말 얻기

랜덤하게 선택하기

```
from qgis import processing
processing.algorithmHelp("native:buffer")
```

```
...
```

알고리즘 실행하기

이 예시의 경우, 프로젝트에 추가된 임시 메모리 레이어에 결과를 저장합니다.

```
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'}
```

알고리즘이 몇 개나 있는가?

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

제공자가 몇 개나 있는가?

```
from qgis.core import QgsApplication

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

표현식이 몇 개나 있는가?

```
from qgis.core import QgsExpression

len(QgsExpression.Functions())
```

21.10 장식자 (decorator)

저작권 (CopyRight)

```
from qgis.PyQt.Qt import QTextDocument
from qgis.PyQt.QtGui import QFont

mQFont = "Sans Serif"
mQFontsize = 9
mLabelQString = "© QGIS 2019"
mMarginHorizontal = 0
mMarginVertical = 0
mLabelQColor = "#FF0000"

INCHES_TO_MM = 0.0393700787402 # 1 millimeter = 0.0393700787402 inches
case = 2

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
def add_copyright(p, text, xOffset, yOffset):
    p.translate( xOffset , yOffset  )
    text.drawContents(p)
    p.setWorldTransform( p.worldTransform() )

def _on_render_complete(p):
    deviceHeight = p.device().height() # Get paint device height on which this painter is currently painting
    deviceWidth  = p.device().width() # Get paint device width on which this painter is currently painting
    # Create new container for structured rich text
    text = QTextDocument()
    font = QFont()
    font.setFamily(mQFont)
    font.setPointSize(int(mQFontsize))
    text.setDefaultFont(font)
    style = "<style type=\"text/css\"> p {color: " + mLabelQColor + "}</style>"
    text.setHtml( style + "<p>" + mLabelQString + "</p>" )
    # Text Size
    size = text.size()

    # RenderMillimeters
    pixelsInchX  = p.device().logicalDpiX()
    pixelsInchY  = p.device().logicalDpiY()
    xOffset  = pixelsInchX  * INCHES_TO_MM * int(mMarginHorizontal)
    yOffset  = pixelsInchY  * INCHES_TO_MM * int(mMarginVertical)

    # Calculate positions
    if case == 0:
        # Top Left
        add_copyright(p, text, xOffset, yOffset)

    elif case == 1:
        # Bottom Left
        yOffset = deviceHeight - yOffset - size.height()
        add_copyright(p, text, xOffset, yOffset)

    elif case == 2:
        # Top Right
        xOffset  = deviceWidth  - xOffset - size.width()
        add_copyright(p, text, xOffset, yOffset)

    elif case == 3:
        # Bottom Right
        yOffset  = deviceHeight - yOffset - size.height()
        xOffset  = deviceWidth  - xOffset - size.width()
        add_copyright(p, text, xOffset, yOffset)

    elif case == 4:
        # Top Center
        xOffset = deviceWidth / 2
        add_copyright(p, text, xOffset, yOffset)

    else:
        # Bottom Center
        yOffset = deviceHeight - yOffset - size.height()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        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 작성자 (composer)

이름으로 조판 출력하기

```
composerTitle = 'MyComposer' # Name of the composer

project = QgsProject.instance()
projectLayoutManager = project.layoutManager()
layout = projectLayoutManager.layoutByName(composerTitle)
```

21.12 소스

- QGIS Python (PyQGIS) API
- QGIS C++ API
- 스택 오버플로의 QGIS 질문들
- 클라스 칼손 (Klas Karlsson) 의 스크립트