PyQGIS developer cookbook

Реліз 2.6

QGIS Project

May 22, 2015

Зміст

1 Вступ **1**
- 1.1 Консоль Python . . . 1
- 1.2 Плаґіни на Python . . . 2
- 1.3 Програми на Python . . . 2

2 Завантаження шарів **5**
- 2.1 Векторні шари . . . 5
- 2.2 Растрові шари . . . 6
- 2.3 Реєстр шарів карти . . . 7

3 Робота з растровими шарами **9**
- 3.1 Інформація про шар . . . 9
- 3.2 Стиль відображення . . . 9
- 3.3 Оновлення шарів . . . 11
- 3.4 Отримання значень . . . 11

4 Робота з векторними шарами **13**
- 4.1 Перегляд об'єктів векторного шару . . . 13
- 4.2 Редагування векторних шарів . . . 14
- 4.3 Редагування векторних шарів з використанням буфера змін . . . 15
- 4.4 Використання просторового індексу . . . 16
- 4.5 Збереження векторних шарів . . . 17
- 4.6 Memory провайдер . . . 18
- 4.7 Зовнішній вигляд (стиль) векторних шарів . . . 19
- 4.8 Інші теми . . . 26

5 Робота з геометрією **27**
- 5.1 Створення геометрії . . . 27
- 5.2 Доступ до геометрії . . . 27
- 5.3 Геометричні предикати та операції . . . 28

6 Робота з проекціями **31**
- 6.1 Системи координат . . . 31
- 6.2 Проекції . . . 32

7 Робота з картою **33**
- 7.1 Вкладення карти . . . 33
- 7.2 Використання інструментів карти . . . 34
- 7.3 Гумові полоси та маркери вершин . . . 35
- 7.4 Створення власних інструментів карти . . . 36
- 7.5 Створення власних елементів карти . . . 37

8 Рендерінг карти та друк **39**

8.1 Просте відображення . . . 39
8.2 Відображення за допомогою макетів . . . 40

9 Вирази, фільтрація та обчислення значень **43**
9.1 Аналіз виразів . . . 44
9.2 Обчислення виразів . . . 44
9.3 Приклади . . . 45

10 Читання за збереження настройок **47**

11 Взаємодія з користувачем **49**
11.1 Повідомлення. Клас `QgsMessageBar` . . . 49
11.2 Індикація прогресу . . . 50
11.3 Реєстрація помилок . . . 51

12 Розробка плаґінів на Python **53**
12.1 Розробка плаґіна . . . 53
12.2 Файли плаґіна . . . 54
12.3 Документація . . . 58

13 Настройка IDE для розробки плаґінів **59**
13.1 Про настройку IDE у Windows . . . 59
13.2 Зневадження в Eclipse та PyDev . . . 60
13.3 Зневадження з PDB . . . 64

14 Шари плаґінів **65**
14.1 Успадкування `QgsPluginLayer` . . . 65

15 Сумісність з попередніми версіями QGIS **67**
15.1 Меню плаґіна . . . 67

16 Публікація плаґіна **69**
16.1 Офіційний репозиторій плаґінів . . . 69

17 Фрагменти коду **71**
17.1 Як викликати метод за комбінацією клавіш . . . 71
17.2 Як керувати видимістю шарів . . . 71
17.3 Як отримати доступ до таблиці атрибутів вибраних об'єктів . . . 71

18 Бібліотека аналізу мереж **73**
18.1 Загальна інформація . . . 73
18.2 Побудова графу . . . 73
18.3 Аналіз графу . . . 75

Індекс **81**

Розділ 1

Вступ

Цей документ створювався як підручник та довідник. І хоча в ньому не розглядаються всі можливі варіанти використання, він повинен дати змістовний огляд основного функціоналу.

Починаючи з версії 0.9, в QGIS реалізовано підтримку сценаріїв на мові Python. Ми вибрали саме Python, оскільки це одна з найпоширеніших скриптових мов. Прив'язки (bindings) PyQGIS залежать від SIP та PyQt4. Основною причиною використання SIP замість більш розповсюдженого SWIG є те, що код QGIS залежить від бібліотек Qt. А прив'язки для Qt (PyQt) також генеруються за допомогою SIP, це дозволяє забезпечити прозору інтеграцію PyQGIS та PyQt.

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

Існує декілька способів використання Python-прив'язок QGIS, вони розглядаються у наступних розділах:

- виконання команд у консолі Python QGIS
- створення та використання плаґінів на Python
- створення власних програм з використанням API QGIS

Існує повний опис QGIS API, у якому зібрано інформацію про всі класи бібліотек QGIS. QGIS Python API практично ідентичне C++ API.

Крім того, деяка інформація про розробку з використанням PyQGIS можна знайти у блозі QGIS. Так у дописі QGIS tutorial ported to Python наведено приклади простих автономних програм. Хороший спосіб дізнатися про розробку плаґінів — завантажити існуючі модулі з репозиторію плаґінів та ознайомитися з їх кодом. Не забувайте що у каталозі `python/plugins/` встановленої QGIS, також знаходиться декілька плаґінів, які можна використовувати для навчальних потреб.

1.1 Консоль Python

Для маленьких сценаріїв можна використовувати вбудовану в QGIS консоль Python. Відкривається вона з меню *Плаґіни* → *Консоль Python*. Консоль відкривається як немодальне вікно:

Вищенаведений малюнок показує як отримати вибраний у легенді шар, дізнатися його ідентифікатор та, якщо шар векторний, кількість його об'єктів. Для взаємодії з інтерфейсом QGIS використовується змінна `iface`, яка є екземпляром `QgsInterface`. Через цей інтерфейс можна звертатися до карти, меню, панелей інструментів та інших частин QGIS.

Для зручності користувачів, під час відкриття консолі виконуються такі команди (у майбутньому цей список можна буде розширювати)

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

Тим, хто використовує консоль часто, варто призначити комбінацію клавіш для її виклику (в меню *Налаштування* → *Комбінації клавіш...*)

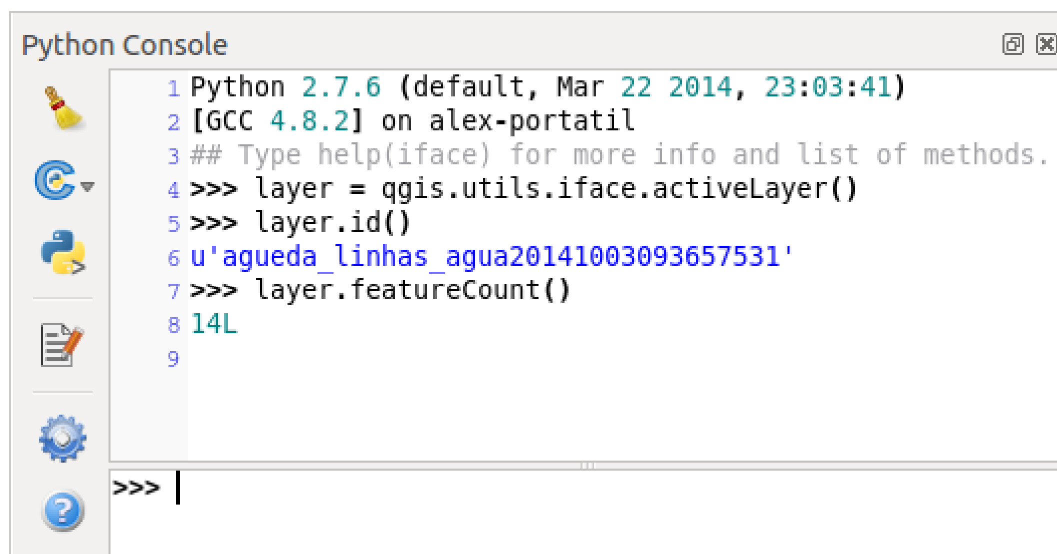

Рис. 1.1: Консоль Python QGIS

1.2 Плаґіни на Python

QGIS дозволяє розширювати свій функціонал через плаґіни. Спочатку це було можливим лише на мові C++. Після реалізації підтримки Python у QGIS, з'явилась можливість використання плаґінів, написаних на Python. Головна перевага таких плаґінів у порівнянні з плаґінами на C++ — простота розповсюдження (відпадає необхідність у компіляції для різних платформ) та розробки.

З моменту введення підтримки Python було розроблено багато плаґінів. Менеджер плаґінів дозволяє легко отримувати, оновлювати та видаляти Python плаґіни. більше інформації про різні джерела плаґінів розміщена на сторінці Python Plugin Repositories.

Створювати плаґіни на Python дуже просто — дивіться розділ *Розробка плаґінів на Python*.

1.3 Програми на Python

Під час обробки ГІС-даних часто набагато зручніше створити декілька сценаріїв для автоматизації процесу ніж постійно виконувати ті самі дії знову і знову. PyQGIS допомагає зробити це — просто імпортуйте модуль `qgis.core`, ініціалізуйте його і у вас все готове до обробки даних.

Або ж вам може знадобитися інтерактивна програма з певним функціоналом ГІС — вимірювання довжин та площ, експорт карти в PDF або щось інше. Модуль `qgis.gui` містить різноманітні елементи інтерфейсу, найголовніший з них — віджет карти, який легко інтегрується у програму та підтримує переміщення, масштабування і будь-які інші інструменти карти.

1.3.1 Використання PyQGIS у програмах

Примітка: *не* використовуйте ім'я `qgis.py`, для своїх сценаріїв — Python не зможе імпортувати прив'язки, оскільки ім'я сценарію буде «затінювати» їх.

Перш за все необхідно імпортувати модуль `qgis.core` та вказати шлях, по якому QGIS буде шукати ресурси — базу даних проекцій, провайдери тощо. Якщо під час встановлення шляхів пошуку другий аргумент задано як `True`, QGIS ініціалізує всі шляхи стандартними значеннями з використанням вказаного префіксу. Виклик функції `initQgis()` дуже важливий, так як він дозволяє QGIS виконати пошук доступних провайдерів.

```
from qgis.core import *

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

# load providers
QgsApplication.initQgis()
```

Тепер можна працювати з API QGIS — завантажувати шари, виконувати обробку даних або створити графічний інтерфейс з картою. Можливості нескінченні :-)

По завершенню роботи з бібліотеками QGIS викличте `exitQgis()` щоб впевнитися, що всі ресурсу звільнено (наприклад, список шарів очищено і всі шари видалені):

```
QgsApplication.exitQgis()
```

1.3.2 Запуск програм

Необхідно вказати системі де саме шукати бібліотеки QGIS та відповідні модулі Python — інакше під час запуску з'явиться повідомлення про помилку:

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

Для цього необхідно встановити змінну оточення `PYTHONPATH`. У наступних командах заміняйте `qgispath` на реальне розміщення QGIS у вашій системі:

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

Тепер розміщення модулів PyQGIS відоме, але вони в свою чергу залежать від бібліотек `qgis_core` та `qgis_gui` (модулі Python лише «обгортки» над ними). У більшості випадків операційна система не знає де знаходяться ці бібліотеки, тому ви знову отримаєте помилку імпорту (повідомлення може відрізнятися в залежності від операційної системи):

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

Для вирішення цієї проблеми додайте каталоги з бібліотеками QGIS до шляхів пошуку динамічного компонувальника:

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

Ці команди можна розмістити у стартовому командному файлі, який і буде налаштовувати систему. Для розгортання автономних програм, що використовують PyQGIS, можна використовувати два способи:

- вимагати від користувача встановлення QGIS перед встановленням вашої програми. Інсталятор програми повинен перевіряти наявність бібліотек QGIS у стандартних каталогах та дозволяти користувачу вказувати їх розміщення, якщо знайти їх автоматично не вдалося. Перевагою цього методу є простота, однак він потребує додаткових дій з боку користувача.
- включати QGIS в інсталятор своєї програми. Підготовка до випиську стане більш складною, а сам інсталятор більш об'ємним. Але користувачі будуть позбавлені необхідності завантажувати та встановлювати додаткові програми самостійно.

Ці два підходи можна комбінувати — розгортати програму разом з QGIS у Windows та Mac OS X, а в Linux залишити встановлення QGIS на розсуд користувача.

Розділ 2

Завантаження шарів

Давайте завантажимо декілька шарів з даними. QGIS розділяє шари на векторні та растрові. Крім того, існують користувацькі типи шарів, але їх обговорення поза межами цього розділу.

2.1 Векторні шари

Щоб завантажити векторний шар необхідно вказати ідентифікатор джерела даних, ім'я шару та назва провайдера даних:

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

Ідентифікатор джерела даних це рядок, специфічний для кожного провайдера даних. Ім'я шару використовується у віджеті списку шарів. Необхідно перевіряти результат завантаження шару. Якщо виникла помилка буде повернений неправильний екземпляр.

Нижче показується як завантажувати шари з різних джерел:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Згенерувати правильний URI можна за допомогою стандартної бібліотеки `urllib`

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

2.2 Растрові шари

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

```
fileName = "/path/to/raster/file.tif"
fileInfo = QFileInfo(fileName)
baseName = fileInfo.baseName()
rlayer = QgsRasterLayer(fileName, baseName)
if not rlayer.isValid():
  print "Layer failed to load!"
```

Також можна завантажувати растрові шари з серверів WCS

```
layer_name = 'elevation'
uri = QgsDataSourceURI()
uri.setParam ('url', 'http://localhost:8080/geoserver/wcs')
```

```
uri.setParam ( "identifier", layer_name)
rlayer = QgsRasterLayer(uri, 'my_wcs_layer', 'wcs')
```

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

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

2.3 Реєстр шарів карти

Якщо ви хочете використовувати відкрити шари для рендерінгу карти — не забудьте додати їх до реєстру шарів. Реєстр шарів карти стане їх власником, а отримати доступ до будь-якого шару можна буде за допомогою унікального ідентифікатора. Коли шар видаляється з реєстру шарів, відбувається його знищення.

Adding a layer to the registry:

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

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

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

TODO: More about map layer registry?

Розділ 3

Робота з растровими шарами

У цьому розділі описано операції, які можна виконувати з растровими шарами.

3.1 Інформація про шар

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

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

3.2 Стиль відображення

Після завантаження растровий шар відображається стилем, що відповідає його типу. Стиль можна змінити у діалозі властивостей растра або програмним шляхом. Існують наступні стилі:

Індекс	Константа: QgsRasterLater.X	Коментар
1	SingleBandGray	Одноканальне зображення у відтінках сірого
2	SingleBandPseudoColor	Одноканальне зображення, відображається з використанням псевдокольору
3	PalettedColor	Шар з «палітрою», відображається з використанням таблиці кольорів
4	PalettedSingleBandGray	Шар з «палітрою», відображається у відтінках сірого
5	PalettedSingleBandPseudoColor	Шар з «палітрою», відображається з використанням псевдокольору
7	MultiBandSingleBandGray	Шар складається з 2 або більше каналів, відображається лише один канал у відтінках сірого
8	MultiBandSingleBandPseudoColor	Шар складається з 2 або більше каналів, відображається лише один канал з використанням псевдокольору
9	MultiBandColor	Шар складається з 2 або більше каналів, відображення у відповідності з кольорами простору RGB

To query the current drawing style:

```
>>> rlayer.drawingStyle()
9
```

Одноканальні растри можуть відображатися або у відтінках сірого (менші значення = чорний, більші значення = білий) або з використанням псевдокольору, коли значенням растра призначається свій колір. Крім того, одноканальні растри з палітрою можуть відображатися у відповідності до палітри. Багатоканальні растри, як правило, відображаються шляхом встановлення відповідності між їх каналами та кольорами RGB. Ще один спосіб — використовувати лише один канал для відображення у відтінках сірого або з використанням псевдокольору.

У наступних розділах описано як визначити та змінити стиль відображення шару. Після внесення змін необхідно буде оновити карту, див. *Оновлення шарів*.

TODO: contrast enhancements, transparency (no data), user defined min/max, band statistics

3.2.1 Одноканальні растри

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

```
>>> rlayer.setDrawingStyle(QgsRasterLayer.SingleBandPseudoColor)
>>> rlayer.setColorShadingAlgorithm(QgsRasterLayer.PseudoColorShader)
```

The `PseudoColorShader` is a basic shader that highlights low values in blue and high values in red. Another, `FreakOutShader` uses more fancy colors and according to the documentation, it will frighten your granny and make your dogs howl.

There is also `ColorRampShader` which maps the colors as specified by its color map. It has three modes of interpolation of values:

- лінійна (`INTERPOLATED`): кінцевий колір є результатом лінійної інтерполяції кольорів
- дискретна (`DISCRETE`): використовується колір що відповідає або більший за колір відповідного значення карти кольорів
- точна (`EXACT`): інтерполяція відсутня, відображаються лише пікселі за значеннями, які є у карті кольорів

Створити градієнт від зеленого до жовтого (для значенњ від 0 до 255) можна так:

```
>>> rlayer.setColorShadingAlgorithm(QgsRasterLayer.ColorRampShader)
>>> lst = [ QgsColorRampShader.ColorRampItem(0, QColor(0,255,0)), \
    QgsColorRampShader.ColorRampItem(255, QColor(255,255,0)) ]
>>> fcn = rlayer.rasterShader().rasterShaderFunction()
```

```
>>> fcn.setColorRampType(QgsColorRampShader.INTERPOLATED)
>>> fcn.setColorRampItemList(lst)
```

To return back to default gray levels, use:

```
>>> rlayer.setDrawingStyle(QgsRasterLayer.SingleBandGray)
```

3.2.2 Багатоканальні растри

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

```
>>> rlayer.setGreenBandName(rlayer.bandName(1))
>>> rlayer.setRedBandName(rlayer.bandName(2))
```

In case only one band is necessary for visualization of the raster, single band drawing can be chosen — either gray levels or pseudocolor, see previous section:

```
>>> rlayer.setDrawingStyle(QgsRasterLayer.MultiBandSingleBandPseudoColor)
>>> rlayer.setGrayBandName(rlayer.bandName(1))
>>> rlayer.setColorShadingAlgorithm(QgsRasterLayer.PseudoColorShader)
>>> # now set the shader
```

3.3 Оновлення шарів

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

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

Перша конструкція потрібна щоб впевнитися, що при використання кешу рендерера закешовані зображення шару видалені. Цей функціонал доступний починаючи з QGIS 1.4, у попередніх версіях QGIS ця функція відсутня — тому, перед цим, щоб бути впевненим у працездатності коду в будь-якій версії QGIS, робиться перевірка на наявність метода.

Друга конструкція відправляє сигнал, який змушує оновитися всі карти, що містять шар.

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

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

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

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

3.4 Отримання значень

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

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

В нашому випадку метод `results()` поверне словник, з номерами каналі в якості ключів, та значеннями растра в якості значень.

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

Розділ 4

Робота з векторними шарами

Цей розділ описує операції які можна виконувати з векторними шарами.

4.1 Перегляд об'єктів векторного шару

Перегляд об'єктів векторного шару — одна з найчастіших операцій. Нижче показано простий код для цієї вирішення задачі, а також для відображення деякої інформації про кожний об'єкт. Вважається, що змінна `layer` містить об'єкт `QgsVectorLayer`.

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

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

    # fetch attributes
    attrs = feature.attributes()

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

Attributes can be referred by index.

```
idx = layer.fieldNameIndex('name')
print feature.attributes()[idx]
```

4.1.1 Перегляд вибраних об'єктів

Convenience methods.

For the above cases, and in case you need to consider selection in a vector layer in case it exist, you can use the `features()` method from the built-in Processing plugin, as follows:

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

This will iterate over all the features in the layer, in case there is no selection, or over the selected features otherwise.

if you only need selected features, you can use the :func: *selectedFeatures* method from vector layer:

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

4.1.2 Перегляд певної множини об'єктів

Якщо необхідно переглянути лише певну множину об'єктів шару, наприклад, об'єкти, що попадають у задану область, слід додати параметр `QgsFeatureRequest` у виклик методу `getFeatures()`. Приклад

```
request=QgsFeatureRequest()
request.setFilterRect(areaOfInterest)
for f in layer.getFeatures(request):
    ...
```

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

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

4.2 Редагування векторних шарів

Більшість провайдерів векторних даних підтримує редагування. Іноді вони дозволяють виконували лише деякі операції редагування. Отримати список доступних операцій можна за допомогою метода `capabilities()`

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

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

4.2.1 Створення об'єктів

Створіть декілька екземплярів `QgsFeature` та передайте список цих об'єктів у метод `addFeatures()` провайдера. Провайдер поверне два значення: результат операції (`True`/`False`) та список створених

об'єктів (їх ID призначаються сховищем даних)

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

4.2.2 Видалення об'єктів

Для видалення об'єктів достатньо передати список їх ідентифікаторів

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

4.2.3 Редагування об'єктів

Можна редагувати як геометрію об'єкта, так і його атрибути. Наступний приклад спочатку модифікує значення атрибутів з індексами 0 та 1, а потім модифікує геометрію

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

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

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

4.2.4 Створення та видалення полів

Щоб створити поля (атрибути), необхідно створити список з описом полів. Для видалення полів достатньо надати список з їх індексами

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

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

After adding or removing fields in the data provider the layer's fields need to be updated because the changes are not automatically propagated.

```
layer.updateFields()
```

4.3 Редагування векторних шарів з використанням буфера змін

Під час редагування векторних даних у QGIS, спочатку необхідно перевести шар у режим редагування, потім внести зміни, і, нарешті, зафіксувати (або скасувати) ці зміни. Всі зміни, які ви зробили, не мають сили поки їх не буде зафіксовано — вони зберігаються у буфері змін шару. Цю можливість можна використовувати і програмно — це ще один спосіб редагування шарів, який доповнює прямий доступ до даних через провайдер. Користуватися цим методом слід тоді, коли користувачу надаються графічні інструменти редагування, щоб він міг вирішувати приймати

результат редагування чи ні, а також мав можливість використовувати інструменти повтора та скасування. Під час фіксації змін все операції з буфера змін будуть передані провайдеру.

Дізнатися чи знаходиться шар у режимі редагування можна за допомогою метода `isEditing()` — функції редагування працюють лише в режимі редагування. Використання операцій редагування показано нижче

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

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

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

Для того, щоб операції повтору/скасування працювали правильно, описані вище методи повинні бути розміщені всередині пакета змін. (Якщо вам не потрібен функціонал повтору/скасування змін і треба зберігати зміни негайно, все зводиться до *редагування через провайдер*). Ось приклад використання можливості скасування змін

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

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

if problem_occurred:
  layer.destroyEditCommand()
  return

# ... more editing ...

layer.endEditCommand()
```

Метод `beginEndCommand()` створює внутрішню «активну» команду та записує всі зміни у векторному шарі. Після виклику `endEditCommand()` ця команда буде розміщена у стеку скасування і користувач зможе скасувати або повторити її через GUI. Якщо в процесі редагування трапилась помилка, метод `destroyEditCommand()` видалить команду та скасує всі зроблені зміни, що сталися з моменту активації цієї команди.

Для активації режиму редагування призначений метод `startEditing()`, за завершення редагування відповідають методи `commitChanges()` та `rollback()` — але в загальному випадку ці методи вам не знадобляться, оскільки їх викликати повинен користувач.

4.4 Використання просторового індексу

Просторовий індекс може значно збільшити продуктивність вашого коду, якщо він часто виконує операції читання векторного шару. Уявіть наприклад, що ви реалізуєте алгоритм інтерполяції і для заданої точки необхідно знайти 10 найближчих об'єктів точкового шару аби використати їх для обчислення інтерпольованого значення. Без просторового індексу єдиний спосіб зробити це в QGIS — обчислити відстані від всіх точок до заданої а потім порівняти їх між собою. Це може бути досить тривалою операцією, особливо якщо її необхідно повторити для декількох точок. Набагато ефективніше цю задачу можна вирішити за допомогою просторового індексу.

Шар без просторового індексу можна порівняти з телефонним довідником, у якому телефонні номери не відсортовані або не впорядковані якимось чином. Єдиний спосіб знайти потрібний номер

в такому випадку — переглядати довідник сторінка за сторінкою, поки потрібна інформація не буде знайдена.

QGIS не створює просторові індекси для векторних шарів автоматично, це залишено на розсуд користувача. Ось необхідні етапи.

1. створення просторового індексу — наступний код створить пустий індекс

   ```
   index = QgsSpatialIndex()
   ```

2. додати об'єкти до індексу — індекс приймає об'єкт `QgsFeature` та розміщує його у внутрішній структурі даних. Об'єкт можна створити вручну або використовувати об'єкти, отримані в результаті роботи викликів `nextFeature()` провайдера

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

3. після заповнення індексу можна переходити до виконання запитів

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

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

4.5 Збереження векторних шарів

Для збереження векторних шарів використовується класс `QgsVectorFileWriter`. Він дозволяє створювати файли у будь-якому OGR-сумісному форматі (shape-файли, GeoJSON, KML та інші).

Існує два способи збереження векторних даних:

- з екземпляра `QgsVectorLayer`

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

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

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

  Третій параметр задає кодування тексту. Він необхідний для нормальної роботи деяких драйверів (зокрема, драйверу shape-файлів). Але якщо ви не використовуєте міжнародні символи, спеціально турбуватися про правильне кодування не треба. Четвертий параметр, який ми залишили пустим (`None`), задає вихідну систему координат — якщо передається правильний екземпляр `QgsCoordinateReferenceSystem`, шар буде трансформовано у відповідну систему координат.

  Правильні імена драйверів можна переглянути на сторінці supported formats by OGR — в якості ім'я драйвера вказано у стовпці «Code». У разі необхідності можна експортувати лише вибрані об'єкти, вказати додаткові параметри драйвера або заборонити експорт атрибутів — за детальною інформацією звертайтесь до документації.

- з окремих об'єктів

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

  # create an instance of vector file writer, which will create the vector file.
  # Arguments:
  # 1. path to new file (will fail if exists already)
  ```

```
# 2. encoding of the attributes
# 3. field map
# 4. geometry type - from WKBTYPE enum
# 5. layer's spatial reference (instance of
#    QgsCoordinateReferenceSystem) - optional
# 6. driver name for the output file
writer = QgsVectorFileWriter("my_shapes.shp", "CP1250", fields, QGis.WKBPoint, None, "ESRI Shapefile")

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

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

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

4.6 Memory провайдер

Memory провайдер здебільшого призначений для використання розробниками програм та плагінів. Він не записує дані на диск, що дозволяє розробникам використовувати його в якості швидкого сховища тимчасових даних.

Провайдер дозволяє створювати текстові, цілі та десяткові поля.

Memory провайдер також підтримує просторове індексування, індекс можна побудувати викликом методу `createSpatialIndex()` провайдера. Після створення індексу перегляд об'єктів у межах невеликих регіонів стане значно швидшим (оскільки будуть запитувати лише об'єкти, що попадають у заданий прямокутник).

Щоб створити тимчасовий шар за допомогою memory провадера достатньо вказати `"memory"` в якості ідентифікатора провайдера у конструкторі `QgsVectorLayer`.

У конструктор також передається URI, що задає тип геометрії шару. Це може бути: `"Point"`, `"LineString"`, `"Polygon"`, `"MultiPoint"`, `"MultiLineString"` або `"MultiPolygon"`.

URI також може містити інформацію про систему координат, поля, та настройки просторового індексування. Використовується наступний синтаксис:

crs=definition Задає систему координат шару, в якості `definition` допускається використання будь-якого формату, що приймається `QgsCoordinateReferenceSystem.createFromString()`

index=yes Вказує чи буде провайдер використовувати просторовий індекс

field=name:type(length,precision) Описує атрибути шару. Кожний атрибут має ім'я та, необов'язково, тип (цілий, з плаваючою комою або текст), довжину так точність. Може бути декілька таких описів.

Нижче показано URI, який містить всі ці параметри

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

Наступний фрагмент коду демонструє створення та заповнення даними memory провайдера

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

# add fields
pr.addAttributes([QgsField("name", QVariant.String),
```

```
                    QgsField("age",  QVariant.Int),
                    QgsField("size", QVariant.Double)])

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

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

Нарешті, перевіримо чи все пройшло успішно

```
# show some stats
print "fields:", len(pr.fields())
print "features:", pr.featureCount()
e = layer.extent()
print "extent:", e.xMin(),e.yMin(),e.xMax(),e.yMax()

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

4.7 Зовнішній вигляд (стиль) векторних шарів

Під час візуалізації векторного шару, зовнішній вигляд даних визначається **рендерером** та **символами**, які асоційовані з шаром. Символи це класи, які займаються візуалізацією об'єктів, а рендерер визначає який саме символ буде використовуватися для певного об'єкта.

Отримати ренедерер шару можна так:

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

Тепер можна переглянути інформацію про рендерер

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

У бібліотеці ядра QGIS реалізовано декілька рендерерів:

Тип	Клас	Коментар
singleSymbol	QgsSingleSymbolRendererV2	Візуалізує всі об'єкти одним і тим же символом
categorizedSymbol	QgsCategorizedSymbolRendererV2	Візуалізує об'єкти з використанням різних символів для кожної категорії
graduatedSymbol	QgsGraduatedSymbolRendererV2	Візуалізує об'єкти з використанням різних символів для кожного діапазону значень

There might be also some custom renderer types, so never make an assumption there are just these types. You can query `QgsRendererV2Registry` singleton to find out currently available renderers.

Існує можливість отримати дамп вмісту рендерера у текстовому вигляді — це може бути корисним під час зневадження

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

4.7.1 Простий знак

Отримати символ, що використовується для візуалізації можна за допомогою метода `symbol()`, а для його модифікації служить метод `setSymbol()` (примітка для розробників на C++: рендерер стає власником символа).

4.7.2 Рендерер категоріями

Дізнатися та встановити ім'я атрибута, який буде використовуватися для класифікації можна за допомогою методів `classAttribute()` та `setClassAttribute()`.

А так отримують список категорій

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

Тут `value()` — величина, що використовується для розрізняння категорій, `label()` — мітка категорії, а метод `symbol()` повертає відповідний символ.

Також рендерер, як правило, зберігає вихідний символ та кольорову шкалу, які використовувалися для класифікації. Отримати їх можна за допомогою методів `sourceColorRamp()` та `sourceSymbol()`.

4.7.3 Градуйований знак

Цей рендерер дуже схожий на градуйований знак, описаний вище, але замість одного значення для класу він оперує діапазном значень, і як наслідок може використовуватися лише з числовими атрибутами.

Отримати інформацію про діапазони, що використовуються, можна так

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

Як і у попередньому випадку доступні методи `classAttribute()` для отримання імені атрибута класифікації, `sourceSymbol()` та `sourceColorRamp()` для отримання вихідного символа та кольорової шкали. Крім того, додатковий метод `mode()` дозволяє дізнатися який алгоритм використовувася для створення діапазонів: рівні інтервали, квантилі або щось інше.

Якщо ви хочете створити свій рендерер категоріями, можете взяти за основу наступний код. Тут показано простий поділ об'єктів на два класи

```
from qgis.core import *

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

```
#now make another symbol and range...
myMin = 50.1
myMax = 100
myLabel = 'Group 2'
myColour = QtGui.QColor('#00eeff')
mySymbol2 = QgsSymbolV2.defaultSymbol(
     myVectorLayer.geometryType())
mySymbol2.setColor(myColour)
mySymbol2.setAlpha(myOpacity)
myRange2 = QgsRendererRangeV2(myMin, myMax, mySymbol2 myLabel)
myRangeList.append(myRange2)
myRenderer = QgsGraduatedSymbolRendererV2('', myRangeList)
myRenderer.setMode(QgsGraduatedSymbolRendererV2.EqualInterval)
myRenderer.setClassAttribute(myTargetField)

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

4.7.4 Робота з символами

Символи представляються базовим класом `QgsSymbolV2` та трьома нащадками:

- `QgsMarkerSymbolV2` — для точок
- `QgsLineSymbolV2` — для ліній
- `QgsFillSymbolV2` — для полігонів

Кожний символ складається з одного чи декількох символьних шарів (похідні класи від `QgsSymbolLayerV2`). Всю роботу з візуалізації виконують саме символьні шари, символ лише контейнер для них.

Отримавши екземпляр символа (наприклад, від рендерера), можна вивчити його. Метод `type()` розкаже маркер це, чи лінія або полігон. Метод `dump()` поверне короткий опис символа. Отримати список шарів символа можна так

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

Дізнатися про колір символа допоможе метод `color()`, а для зміни кольору використовується `setColor()`. У символів типу маркер додатково присутні методи `size()` та `angle()`, які дозволяють отримати інформацію про розмір символа та його кут повороту. А лінійні символи мають метод `width()`, який повертає товщину лінії.

Розмір та товщина за замовчанням задаються у міліметрах, а кут повороту — у градусах.

Робота з символьними шарами

Як уже було сказано, шари символу (похідні від `QgsSymbolLayerV2`) визначають зовнішній вигляд об'єктів. Існує декілька базових класів символьних шарів. Крім того, можна створювати свої символьні шари і таким чином впливати на візуалізацію об'єктів у широких межах. Метод `layerType()` однозначно ідентифікує клас символьного шару — основними і доступними за замовчанням є символьні шари `SimpleMarker`, `SimpleLine` and `SimpleFill`.

Отримати повний список символьних шарів, які можна використовувати у заданому символьному шарі можна так

```
from qgis.core import QgsSymbolLayerV2Registry
myRegistry = QgsSymbolLayerV2Registry.instance()
myMetadata = myRegistry.symbolLayerMetadata("SimpleFill")
```

```
for item in myRegistry.symbolLayersForType(QgsSymbolV2.Marker):
  print item
```

Результат

```
EllipseMarker
FontMarker
SimpleMarker
SvgMarker
VectorField
```

Клас `QgsSymbolLayerV2Registry` управляє базою даних всіх доступних символьних шарів.

Отримати доступ до даних шару можна за допомогою методу `properties()`, який поверне словник (пари ключ-значення) характеристик, що впливають на зовнішній вигляд. Крім того, існують, спільні для всіх типів, методи `color()`, `size()`, `angle()`, `width()` та відповідні модифікатори. Слід пам'ятати, що кут повороту та розмір доступні тільки для символьних шарів типу маркер, а товщина — тільки для символьних шарів типу лінія.

Власні символьні шари

Уявіть, що вам необхідно налаштувати процес візуалізації своїх даних. Ви можете створити власний клас символьного шару, який буде відображати об'єкти саме так, як вам потрібно. Ось приклад маркера, який малює червоні кола заданого радіуса

```
class FooSymbolLayer(QgsMarkerSymbolLayerV2):

  def __init__(self, radius=4.0):
    QgsMarkerSymbolLayerV2.__init__(self)
    self.radius = radius
    self.color = QColor(255,0,0)

  def layerType(self):
    return "FooMarker"

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

  def startRender(self, context):
    pass

  def stopRender(self, context):
    pass

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

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

Метод `layerType()` задає ім'я символьного шару, яке повинно бути унікальним серед всіх символьних шарів. Щоб всі атрибути залишалися незмінними використовуються характеристики. Метод `clone()` повинен повертати копію символьного шару з точно таким ж атрибутами. І нарешті, методи візуалізації: `startRender()` викликається перед візуалізаціює першого об'єкту, а `stopRender()` — по завершенню візуалізації. За власне візуалізацію відповідає метод `renderPoint()`. Координати точки (точок) повинні бути заздалегідь сконвертованими у вихідні координати.

Для поліліній та полігонів єдина відмінність буде у методі візуалізації: слід використовувати

renderPolyline(), якому передається список ліній, або renderPolygon(), якому передається список точок, що утворюють зовнішню межу, та список внутрішніх кілец (або None) другим параметром.

Гарною практикою є реалізація інтерфейсу для настройки атрибутів символьного шару, що дозволяє користувачам налаштовувати зовнішній вигляд. Так, у нашому прикладі можна надати користувачам можливість змінювати радіус кола. Реалізувати це можна так

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

    self.layer = None

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

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

  def symbolLayer(self):
    return self.layer

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

Цей віджет можна вбудувати у діалог настройки символа. Коли символьний шар вибирається у діалозі настройки символа, створюється екземпляр символьного шару та екземпляр відповідного віджету. Потім викликається метод setSymbolLayer() щоб прив'язати символьний шар до віджету. У цьому методі віджет повинен оновити свій інтерфейс, щоб відобразити значення атрибутів символьного шару. Діалог викликає функцію symbolLayer() щоб отримати змінений символьний шар для подального використання.

Після кожної зміни атрибутів віджет повинен посилати сигнал changed(), щоб діалог настройки міг оновити попердній перегляд символа.

Залишився останній крок: розповісти QGIS про існування цих класів. Для цього достатньо додати символьний шар до відповідного реєстру. Звичайно, можно використовувати символьний шар і без внесення у реєстр, але тоді деякий функціонал буде недоступний. Наприклад: завантаження проектів з додатковими символьними шарами або неможливість редагування атрибутів символьного шару.

Спочатку необхідно створити метадані символьного шару

```
class FooSymbolLayerMetadata(QgsSymbolLayerV2AbstractMetadata):

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

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

```
  def createSymbolLayerWidget(self):
    return FooSymbolLayerWidget()

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

У конструктор батьківського класу необхідно передати тип шару (той самий, що повідомляє шар) та тип символу (маркер/лінія/полігон). `createSymbolLayer()` створює екземпляр символьного шару з атрибутами, яказаними у словнику *props*. (Будьте уважні, ключі є екземплярами `QString`, а не об'єктами "str"). Метод `createSymbolLayerWidget()` повинен повертати віджет настройок цього символьного шару.

Останнім рядком ми включаємо символьний шар у реєстр. На цьому все.

4.7.5 Власні рендерери

Можливість створити власний рендерер може стати у нагоді, якщо необхідно реалізувати особливі правила відбору символів для візуалізації. Прикладами таких ситуацій можуть бути: символ повинен відображатися в залежності від значень декількох полів, розмір символа повинен залежати від поточного масштабу тощо.

Наступний код демонструє простий рендерер, який створює два маркери та випадковим чином вибирає одни з них для візуалізації кожного об'єкта

```
import random

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

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

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

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

  def usedAttributes(self):
    return []

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

У конструктор батьківського класу `QgsFeatureRendererV2` необхідно передати ім'я рендерера (повинно бути унікальним). Метод `symbolForFeature()` визначає який символ буде використовуватися для певного об'єкта. `startRender()` та `stopRender()` виконують ініціалізацію/фіналізацію рендерінга символа. Метод `usedAttributes()` може повертати список імен полів, які використовуються рендерером. І нарешті, метод `clone()` повинен повертати копію рендерера.

Як і у випадку символьних шарів, рендерер може мати графічний інтерфейс для настройки параметрів. Він успадковується від класу `QgsRendererV2Widget`. Наступний код створює кнопку, яка дозволяє змінювати один з символів

```
class RandomRendererWidget(QgsRendererV2Widget):
  def __init__(self, layer, style, renderer):
    QgsRendererV2Widget.__init__(self, layer, style)
    if renderer is None or renderer.type() != "RandomRenderer":
      self.r = RandomRenderer()
```

```
    else:
      self.r = renderer
    # setup UI
    self.btn1 = QgsColorButtonV2("Color 1")
    self.btn1.setColor(self.r.syms[0].color())
    self.vbox = QVBoxLayout()
    self.vbox.addWidget(self.btn1)
    self.setLayout(self.vbox)
    self.connect(self.btn1, SIGNAL("clicked()"), self.setColor1)

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

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

У конструктор передається екземпляр активного шару (`QgsVectorLayer`), глобальний стиль (`QgsStyleV2`) та поточний рендерер. Якщо рендерер не задано, або його має інший тип — ми заміннюємо його своїм рендерером, в протилежному випадку використовуємо поточний рендерер (який нам і потрібен). Необхідно оновити віджет, щоб відобразити поточний стан рендерера. При закритті діалога настройок рендерера викликається метод `renderer()` віджета щоб отримати поточний рендерер — він буде підключений до шару.

Залишилось підготувати метадані рендерера та внести його у реєстр інакше завантажити шари з цим рендерером не вдасться, а користувач не побачить його у списку доступних рендерерів. Завершуємо наш приклад з RandomRenderer

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

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

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

Як і у випадку з символьними шарами, абстрактний конструктор метаданих повинен отримати ім'я рендерера, видиме ім'я рендерера (використовується в GUI) та, за бажанням, шлях до іконки рендерера. В метод `createRenderer()` передається екземпляр `QDomElement`, який може використовуватися для відновлення стану рендерера з дерева DOM. Метод `createRendererWidget()` відповідає за створенняє віджета настройки рендерера. Якщо рендерер не має віджета настройки, цей метод може бути відсутнім або просто повертати *None*.

Встановити іконку рендерера можна, передавши її у конструктор `QgsRendererV2AbstractMetadata` третім (необов'язковим) параметром — конструтор базового класу в функціх func:__*init*__ класу `RandomRendererMetadata` буде мати вигляд

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

Іконку можна призначити і пізніше за допомогою метода `setIcon()` класа метаданих. Іконка завантажується або з файлу (як показано вище) або з ресурсів Qt (у складі PyQt4 є компілятор ресурсів `.qrc` для Python).

4.8 Інші теми

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

Розділ 5

Робота з геометрією

Точки, лінії та полігони, які представляють просторові об'єкти, зазвичай називають геометрією. В QGIS вони описуються класом `QgsGeometry`. З усіма можливими типами геометрій можна ознайомитися на сторінці обговорення JTS.

Іноді одна геометрія насправді є колекцією простих (single-part) геометрій. Такі геометрії називаються складеними (multi-part). Якщо складена геометрія містить прості геометрії одного типу, то її називають мульти-точка, мульти-лінія або мульти-полігон. Наприклад, країна, що складається з декількох островів може бути представлена як мульти-полігон.

Координати, що описують геометрії, можуть бути в будь-якій системі координат (CRS). Коли виконується доступ до об'єктів шару, асоційовані геометрії будуть мати систему координат шару.

5.1 Створення геометрії

Існує декілька способів створити геометрію:

- from coordinates

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

  Координати задаються за допомогою класу `QgsPoint`.

  Полілінія описується масивом точок. Полігон в свою чергу описується як список лінійних кілець (тобто замкнених ліній). Перше кільце — зовнішнє (межа), всі наступні необов'язкові кільця описують дірки в полігоні.

  Складені геометрії мають ще один рівень вкладеності: мульти-точка це список точок, мультилінія — список ліній, а мульти-полігон, відповідно, список полігонів.

- from well-known text (WKT)

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

- from well-known binary (WKB)

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

5.2 Доступ до геометрії

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

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

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

Для витягнення інформації з геометрії передбачено спеціальні функції доступу для кожного типу геометрії. Нижче показано як їх використовувати

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

Примітка: кортеж (x, y) насправді не кортеж, а об'єкт класу `QgsPoint`. Отримати його значення можна за допомогою методів `x()` та `y()`.

Для складених геометрій існують аналогічні методи доступу: `asMultiPoint()`, `asMultiPolyline()`, `asMultiPolygon()`.

5.3 Геометричні предикати та операції

QGIS використовує бібліотеку GEOS для виконання різноманітних операцій з геометріями, таких як геометричні предикати (`contains()`, `intersects()`, ...) та операції (`union()`, `difference()`, ...). Також вона може обчислювати геометричні характеристики, такі як площа (для полігонів) або довжина (для полігонів та ліній).

Нижче наведено маленький приклад, де показаний обхід всіх геометрій шару та обчислення деяких геометричних характеристик відповідних об'єктів.

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

Площа та периметр розраховуються методами класу `QgsGeometry` без врахування системи координат шару. Для більш гнучкого обчислення площі та відстані існує клас `QgsDistanceArea`. Якщо перепроектування вимкнене, розрахунки відбуваються на площині, інакше — на еліпсоїді. Якщо еліпсоїд не задано явно, використовуються параметри WGS 84.

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

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

Багато прикладів використання методів аналізу та перетворення векторних даних можна знайти в коді алгоритмів QGIS. Ось декілька посилань для початку:

Additional information can be found in followinf sources:

- Geometry transformation: Reproject algorithm

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

Розділ 6

Робота з проекціями

6.1 Системи координат

Системи координат (Coordinate reference system, CRS) інкапсулюються класом `QgsCoordinateReferenceSystem`. Екземпляри цього класу можна створити різними способами:

- specify CRS by its ID

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

  QGIS використовує три різних ідентифікатора (ID) для кожної системи координат:

  – `PostgisCrsId` — ідентифікатор, що використовується у базах даних PostGIS

  – `InternalCrsId` — внутрішній ідентифікатор QGIS

  – `EpsgCrsId` — ідентифікатор, призначений консорціумом EPSG

  Якщо не задано другий параметр, за замовчанням використовується PostGIS SRID.

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

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

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

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

Бажано перевіряти успішність створення (тобто виконати пошук у базі даних) системи координат: `isValid()` повинен повернути `True`.

Майте на увазі, що для ініціалізації системи координат QGIS повинна здійснити пошук відповідних значень у внутрішній базі даних `srs.db`. Тому, якщо ви розробляєте автономну програму, необхідно правильно налаштувати шляхи за допомогою `QgsApplication.setPrefixPath()`, інакше база даних не буде знайдена. Якщо ви виконуєте команди у консолі Python QGIS або розробляєте плаґін — не хвилюйтесь, все вже налаштовано.

Отримання інформації про систему координат

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

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

6.2 Проекції

Для перетворення між різними системами координат використовується клас `QgsCoordinateTransform`. Простіше за все створити екземпляри вхідної та вихідної системи координат та ініціалізувати ними екземпляр `QgsCoordinateTransform`. Потім просто виконуйте трансформацію, викликаючи метод `transform()`. За замовчанням виконується пряме перетворення, але також можна робити і зворотнє

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

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

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

Розділ 7

Робота з картою

Віджет «карта» (map canvas) є одним з найважливіших, оскільки саме він відповідає за відображення карти, яка складається з накладених один на одного шарів, та дозволяє взаємодіяти як за картою в цілому, так і з окремими шарами. Віджет завжди відображає частину карти, що задана поточним охопленням. Взаємодія з картою відбувається за допомогою **інструментів карти** (map tools): існують інструменти переміщення, масштабування, визначення об'єктів, виміру, редагування векторних шарів тощо. Як і в інших програмах, у кожний момент часу активним може бути лише один інструмент, за необхідності користувач переключається між ними.

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

Кожного разу, коли карта була зсунута, збільшена чи зменшена (або користувач виконує будь-яку іншу дію, яка ініціює оновлення), виконується візуалізація карти в межах поточного охоплення. Шари візуалізуються у зображення (за це відповідає клас `QgsMapRenderer`) і саме це зображення відображається на карті. Графічний елемент (у термінах фреймворку Qt Graphics View), що відповідає за відображення карти, є клас `QgsMapCanvasItem`. Цей клас також відповідає за оновлення візуалізованої карти. Окрім цього елемента, який використовується як фон, може існувати довільна кількість додаткових **елементів карти**. Типовими елементами карти є так звані «гумові» нитки (використовуються під час вимірювань та редагування) або маркери вершин. Елементи карти зазвичай використовуються для відображення роботи інструментів карти. Наприклад, під час створення нового полігону, інструмент карти генерує «гумову» нитку, що показує поточну форму полігону. Всі елементи карти є підкласом `QgsMapCanvasItem`, який розширяє можливості базового об'єкту `QGraphicsItem`.

Таким чином, архітектура карти базується на трьох концепціях:

- карта — для візуалізації даних
- елементи карти — додаткові елементи, що відображаються на карті
- інструменти карти — для взаємодії з картою

7.1 Вкладення карти

Оскільки карта це такий же віджет як і будь-який інший елемент інтерфейсу Qt, її використання, створення та відображення дуже просте

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

Цей код створить нове вікно з картою. Карту також можна вкласти в наявний віджет чи вікно. При використанні Qt Designer та файлів .ui зручно використовувати наступний підхід: на формі розмістити `QWidget`, та перетворити його у новий клас встановивши ім'я класу в `QgsMapCanvas`

та вказавши в якості заголовного файлу `qgis.gui`. Все інше зробить програма `pyuic4`. Це дуже зручний спосіб вкладання карти. Інший спосіб — створити карту та інші елементи інтерфейсу динамічно (в якості дочірніх об'єктів головного вікна або діалогу) та розмістити їх у вікні.

За замовчанням фон карти чорний, згладжування під час візуалізації відсутнє. Щоб зробити фон білим та увімкнути згладжування використовується код

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

(якщо вам цікаво, `Qt` знаходиться у модулі `PyQt4.QtCore`, а `Qt.white` це один з наперед заданих екземплярів класу `QColor`).

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

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

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

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

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

Після виконання цих команд на карті повинен відобразитися завантажений шар.

7.2 Використання інструментів карти

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

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

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

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

    self.canvas.setExtent(layer.extent())
    self.canvas.setLayerSet( [ QgsMapCanvasLayer(layer) ] )

    self.setCentralWidget(self.canvas)

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

    actionZoomIn.setCheckable(True)
```

```
    actionZoomOut.setCheckable(True)
    actionPan.setCheckable(True)

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

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

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

    self.pan()

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

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

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

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

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

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

7.3 Гумові полоси та маркери вершин

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

Створити полілінію можна так

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

Відобразити полігон

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

Зверніть увагу, вершини полігону представлені не простим списком, це список меж полігону: перше кільце є зовнішньою межею, всі інші (необов'язкові) кільця відповідають діркам у полігоні.

Гумові полоси можна налаштовувати, а саме змінювати їх колір та товщину

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

Елементи карти прив'язуються до графічної сцени. Щоб тимчасово сховати їх (а потім знову показати) використовуються методи `hide()` та `show()`. Щоб остаточно видалити елемент необхідно видалити його з графічної сцени

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

(при використанні C++ достатньо просто видалити елемент, однак у Python `del r` видалить лише посилання, а сам об'єкт залишиться в пам'яті, оскільки його власником є карта)

Гумові полоси також можна використовувати для відображення точок, однак для цього краще користуватися спеціальним класом `QgsVertexMarker` (`QgsRubberBand` може намалювати лише прямокутник навколо вказаної точки). Маркер ствоюється так

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

Наступний фрагмент коду показує як відобразити червоний хрестик у позиції [0,0]. За бажанням можна змінити іконку, розмір, колір та товщини пера

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

Тимчасове приховування маркерів та їх повне видалення з карти виконується аналогічно до гумових полос.

7.4 Створення власних інструментів карти

Ви можете створювати власні інструменти карти, щоб реалізувати необхідну реакцію на дії користувача.

Інструменти карти повинні бути нащадками класу `QgsMapTool` або іншого похідного класу. Активація інструмента, як вже було сказано, виконується за допомогою метода `setMapTool()`.

Нижче наведено інструмент карти, який дозволяє вибирати прямокутну область на карті шляхом протягування по карті миші з натиснутою кнопкою. Коли область вказано, координати її вершин виводяться у консоль. Для відображення вибраної області інструмент використовує гумові полоси.

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

  def reset(self):
      self.startPoint = self.endPoint = None
      self.isEmittingPoint = False
      self.rubberBand.reset(QGis.Polygon)

  def canvasPressEvent(self, e):
      self.startPoint = self.toMapCoordinates(e.pos())
```

```
        self.endPoint = self.startPoint
        self.isEmittingPoint = True
        self.showRect(self.startPoint, self.endPoint)

    def canvasReleaseEvent(self, e):
        self.isEmittingPoint = False
        r = self.rectangle()
        if r is not None:
          print "Rectangle:", r.xMin(), r.yMin(), r.xMax(), r.yMax()

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

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

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

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

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

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

        return QgsRectangle(self.startPoint, self.endPoint)

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

7.5 Створення власних елементів карти

TODO: how to create a map canvas item

Розділ 8

Рендерінг карти та друк

Загалом існує два способи отримати друковану карту: швидкий, за допомогою `QgsMapRenderer`, або підготовка макета за допомогою `QgsComposition` та супутніх класів.

8.1 Просте відображення

Відобразити шари за допомогою `QgsMapRenderer` дуже просто — створюється вихідний пристрій (`QImage`, `QPainter` тощо), вказується список шарів, охоплення, розмір вихідного зображення та запускається рендерінг

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

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

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

render = QgsMapRenderer()

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

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

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

# do the rendering
render.render(p)

p.end()

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

8.2 Відображення за допомогою макетів

Редактор макетів стає у нагоді коли вам необхідно отримати щось складніше, ніж дозволяє простий рендерінг, описаний вище. Використовуючи редактор макетів можна створити складний макет, що містить декілька карт, підписи, легенду, таблиці та інші елементи, які ми звичайно бачимо на друкованих картах. Макети можна експортувати у PDF, растрові зображення або роздруковувати напряму.

Редактор макетів складається з багатьох класів. Всі вони знаходяться у бібліотеці ядра. У QGIS існує зручний графічний інтерфейс, який полегшує розміщення елементів, але на жаль, він поки не доступний у бібліотеці графічного інтерфейсу. Якщо ви не знайомі з фремворком‘Qt Graphics View <http://doc.qt.nokia.com/stable/graphicsview.html>‘_, радимо переглянути його документацію зараз, оскільки редактор макетів побудовано на ньому.

Основним класом редактора макетів є `QgsComposition`, який походить від `QGraphicsScene`. Створимо екземпляр цього класу

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

Зверніть увагу, що макет приймає в якості параметра екземпляр `QgsMapRenderer`. Ми припускаємо, зо код буде виконуватися безпосередньо в QGIS, тому використовуємо ренедерер активної карти. Макет використовує різноманітні параметри рендерера, найголовніші з них — список шарів карти та поточне охоплення. Якщо макети використовуються в автономній програмі, вам необхідно створити свій власний екземпляр рендерера, як було показано в попередньому розділі, та передати його до макета.

На макет можна додавати різні елементи (карту, підписи, ...) — ці елементи є похідними від класу `QgsComposerItem`. На сьогодні доступні такі елементи:

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

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

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

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

- legend

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

- scale bar

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

- стрілка
- зображення

- фігура
- таблиця

За замовчанням щойно створені елементи мають нульове положення (лівий верхній куточок сторінки) та нульовий розмір. Положення та розміри завжди задаються у міліметрах

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

Також за замовчанням навколо кожного елемента відображається рамка. Прибрати її можна так

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

Крім створення макетів вручну QGIS має підтримку шаблонів макетів. Шаблони це звичайні макети, збережені у вигляді файлів `.qpt` (формат XML). На жаль, цей функціонал поки недоступний через API.

Після того як макет підготований (всі елементи створено та розміщено в необхідних місцях), можна переходити до генерації вихідного растрового чи векторного файлу.

За замовчанням для виводу використовується аркуш розміру A4 та роздільна здатність 300 DPI. При необхідності ці параметри змінюються. Розмір паперу задається в міліметрах

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

8.2.1 Друк у растр

Наступний фрагмент коду показує як згенерувати растрове представлення макету

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

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

# render the composition
imagePainter = QPainter(image)
sourceArea = QRectF(0, 0, c.paperWidth(), c.paperHeight())
targetArea = QRectF(0, 0, width, height)
c.render(imagePainter, targetArea, sourceArea)
imagePainter.end()

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

8.2.2 Друк у PDF

Наступний фрагмент коду показує як отримати файл формату PDF

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

```
printer.setColorMode(QPrinter.Color)
printer.setResolution(c.printResolution())

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

Розділ 9

Вирази, фільтрація та обчислення значень

QGIS має може виконувати синтаксичний аналіз SQL-подібних виразів. Підтримується лише обмежена підмножина мови SQL. Вирази можуть розглядатися як логічні предикати (обчислюються як `True` чи `False`) або як функції (обчислюються як скалярна величина).

Реалізовано підтримку трьох основних типів даних:

- число — цілі та десяткові числа, наприклад `123`, `3.14`
- рядок — повинні включатися в одинарні лапки: `'hello world'`
- посилання на стовпчик — під час обчислення посилання замінюється на значення вказаного поля. Імена полів не екрануються.

Доступні такі операції:

- арифметичні оператори: `+`, `-`, `*`, `/`, `^`
- дужки: для зміни пріоритету операцій: `(1 + 1) * 3`
- унарні плюс та мінус: `-12`, `+5`
- математичні функції: `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
- геометричні функції: `$area`, `$length`
- функції перетворення типів даних: `to int`, `to real`, `to string`

Крім того, підтримуються наступні предикати:

- порівняння: `=`, `!=`, `>`, `>=`, `<`, `<=`
- відповідність зразку: `LIKE` (з використанням `%` та `_`), `~` (регулярні вирази)
- логічні предикати: `AND`, `OR`, `NOT`
- перевірка на NULL: `IS NULL`, `IS NOT NULL`

Приклади предикатів:

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

Приклади скалярних виразів:

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

9.1 Аналіз виразів

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

9.2 Обчислення виразів

9.2.1 Базові вирази

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

9.2.2 Вирази з об'єктами

У наступних прикладах вираз обчислюється для поточного об'єкта. «Column» це назва поля атрибутивної таблиці шару.

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

Якщо необхідно перевірити декілька об'єктів використовуйте `QgsExpression.prepare()`.Використання :func:'QgsExpression.prepare()'підвищить швидкість аналізу та обчислення виразу.

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

9.2.3 Обробка помилок

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

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

print value
```

9.3 Приклади

Наступний приклад можна використовувати для фільтрації шару та отриманні об'єктів, які задовольняють умові.

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

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

Розділ 10

Читання за збереження настройок

Дуже часто буває необхідно зберегти деякі настройки плаґіна, щоб не змушувати користувача вводити їх знову.

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

Слід розрізняти наступні типи настройок:

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

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

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

  Другий параметр методу `value()` необов'язковий та задає значення за замовчанням, на той випадок, якщо отримати значення з тих чи інших причин не вдасться.

- **настройки проекту** — різні для кожного проекту, тому вони зберігаються безпосередньо у файлі проекту. Прикладом можуть бути колір фону карти або система координат (CRS — в одному проекті може бути білий фон та WGS84, а в іншому —жовтий фон та проекція UTM. Приклад використання нижче

  ```
  proj = QgsProject.instance()

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

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

Як ви бачите, метод `writeEntry()` використовується для всіх типів даних, але для зчитування настройок передбачено окремі методи для кожного типу даних.

- **настройки шару** — ці настройки відносяться до окремих шарів карти. Вони *не* зв'язані з певним джерелом даних, тому якщо з одного shape-файлу створено два шари, вони будуть мати різні настройки. Ці настройки також зберігаються у файлі проекту, тому після відкриття проекту настройки шару будуть відновлені. Цей функціонал було реалізовано в QGIS 1.4. API схоже на API `QSettings` — для читання та запису використовуються екземпляри `QVariant`

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

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

Розділ 11

Взаємодія з користувачем

У цьому розділі розглядаються деякі методи та елементи, які повинні використовуватися для взаємодії з користувачем, щоб дотримуватися однорідності в інтерфейсі.

11.1 Повідомлення. Клас `QgsMessageBar`

З точки зору користувача використання діалогових вікон для повідомлень погана ідея. Для відображення коротких інформаційних повідомлень або попереджень чи повідомлень про помилки краще використовувати панель повідомлень QGIS.

Показати повідомлення в панелі повідомлень QGIS можна за допомогою наступного коду

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

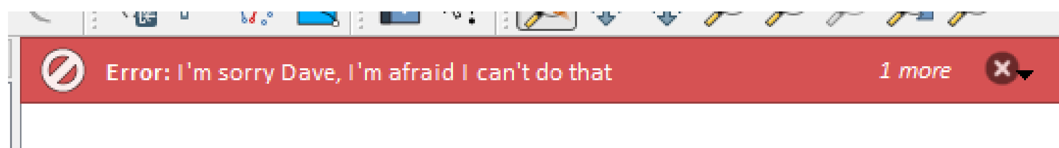

Рис. 11.1: Панель повідомлень QGIS

Можна вказати тривалість відображення

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

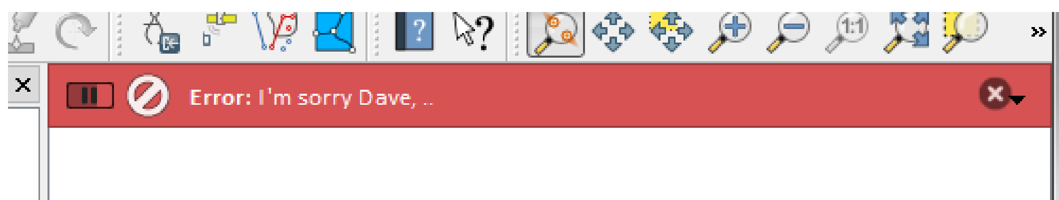

Рис. 11.2: Панель повідомлень QGIS з таймером

Попередні приклади стосувалися повідомлень про помилки, але змінюючи параметр `level` можна створити попередження (`QgsMessageBar.WARNING`) або інформаційне повідомлення (`QgsMessageBar.INFO`).

На панелі повідомлень також можна розмістити додаткові віджети, наприклад, кнопку, яка покаже подробиці

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

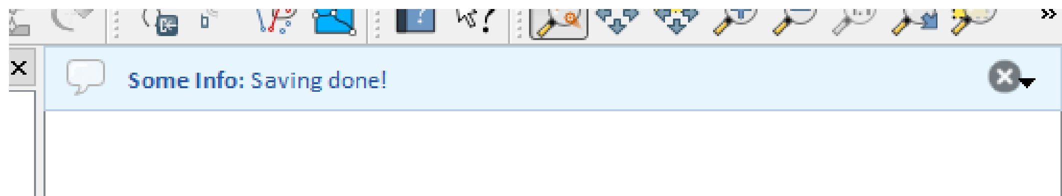

Рис. 11.3: Панель повідомлень QGIS (інформація)

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

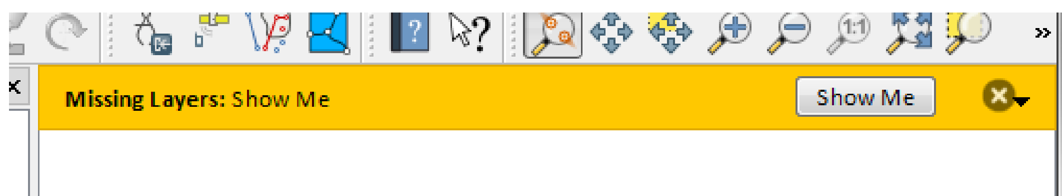

Рис. 11.4: Панель повідомлень QGIS з кнопкою

Панель повідомлень QGIS також можна використовувати у власних діалогових вікнах. Це дозволяє повністю відмовитися від використання діалогових повідомлень.

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

    def run(self):
        self.bar.pushMessage("Hello", "World", level=QgsMessageBar.INFO)
```

11.2 Індикація прогресу

Індикатор прогресу також можна розмістити у панелі повідомлень QGIS, як ми вже бачили, вона дозволяє розміщення віджетів. Ниже наведено приклад, який можна виконати в консолі Python

```
import time
from PyQt4.QtGui import QProgressBar
from PyQt4.QtCore import *
progressMessageBar = iface.messageBar().createMessage("Doing something boring...")
progress = QProgressBar()
progress.setMaximum(10)
progress.setAlignment(Qt.AlignLeft|Qt.AlignVCenter)
progressMessageBar.layout().addWidget(progress)
iface.messageBar().pushWidget(progressMessageBar, iface.messageBar().INFO)
for i in range(10):
```

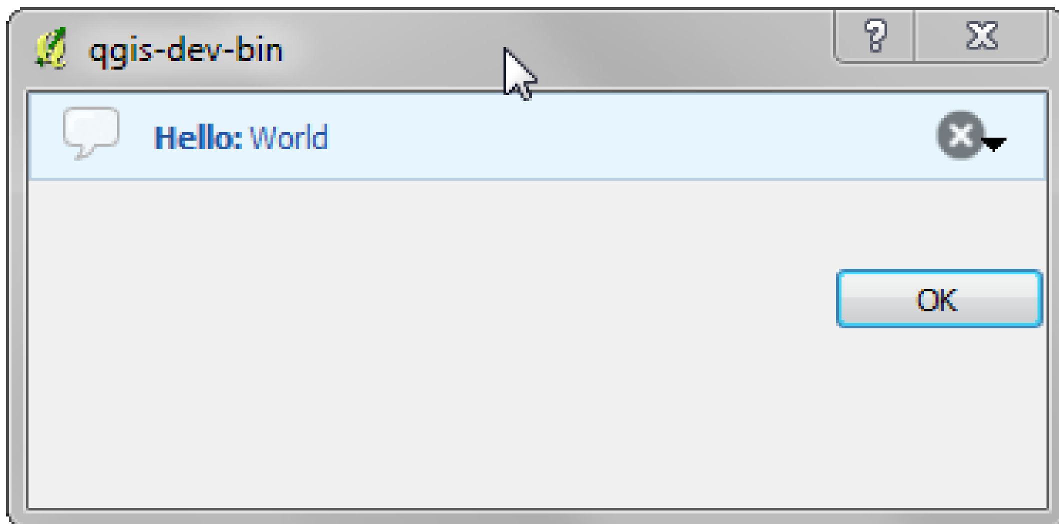

Рис. 11.5: Панель повідомлень QGIS у діалозі

```
    time.sleep(1)
    progress.setValue(i + 1)
iface.messageBar().clearWidgets()
```

Крім того, ви можете використовувати індикатор прогресу у панелі статусу, як це показано в наступному фрагменті коду

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

11.3 Реєстрація помилок

Будь-яку інформацію про виконання коду можна також реєструвати за допомогою системи реєстрації помилок QGIS, як показано нижче

```
QgsMessageLog.logMessage("Your plugin code has been executed correctly", QgsMessageLog.INFO)
QgsMessageLog.logMessage("Your plugin code might have some problems", QgsMessageLog.WARNING)
QgsMessageLog.logMessage("Your plugin code has crashed!", QgsMessageLog.CRITICAL)
```

Розділ 12

Розробка плаґінів на Python

Для розробки плаґінів можна використовувати мову програмування Python. У порівнянні з класичними плаґінами, написаними на C++, їх легше розробляти, розуміти, підтримувати та розповсюджувати через динамічну природу мови Python.

Плаґіни, написані на Python, відображаються разом з плаґінами, написаними на C++, у Менеджері плаґінів. Пошук плаґінів виконується у наступних каталогах:

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

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

Основі етапи:

1. *Ідея*: перш за все необхідна ідея нового плаґіна для QGIS. Навіщо це потрібно? Яку проблеми ви хочете вирішити? Можливо плаґін для цього вже існує?
2. *Створення файлів*: детальніше цей етап описано нижче. Точка входу (`__init.py__`). Заповнення *Метадані плаґіна* (`metadata.txt`). Тіло плагіна (`mainplugin.py`). Форма Qt Designer (`form.ui`) зі своїм `resources.qrc`.
3. *Реалізація*: пишемо код у файлі `mainplugin.py`
4. *Тестування*: Закрийте і знову відкрийте QGIS, завантажте свій модуль. Переконайтесь, що все працює правильно.
5. *Публікація*: опублікуйте свій плаґін у репозиторії плаґінів QGIS або створіть свій власний репозиторій як «арсенал» персональної «ГІС-зброї»

12.1 Розробка плаґіна

З моменту введення підтримки плаґінів на Python у QGIS з'явилось багато плаґінів — на сторінці Plugin Repositories можна знайти деякі з них. Вихідний код цих плаґінів можна використовувати, щоб дізнатися більше про розробку з використанням PyQGIS або для того, щоб переконатися, що розробка не дублюється. Команда розробників QGIS також підтримує *Офіційний репозиторій плаґінів*. Готові до розробки плаґіна, але не маєте ідей? На сторінці Python Plugin Ideas зібрано багато ідей та побажань!

12.1.1 Створення необхідних файлів

Нижче наведено вміст каталогу нашого демонстраційного плаґіна

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

Для чого потрібні ці файли:

- `__init__.py` = точка входу плаґіна. Тут знаходиться метод `classFactory()` та інший код ініціалізації.
- `mainPlugin.py` = основний код плаґіна. Містить інформацію про всі «дії» плаґіна та основний код.
- `resources.qrc` = XML-документ, створений Qt Designer. Тут зберігаються відносні шляхи до ресурсів форм.
- `resources.py` = адаптована для Python версія вищеописаного файлу.
- `form.ui` = графічний інтерфейс (GUI), створений у Qt Designer.
- `form.py` = адаптована для Python версія вищеописаного файлу.
- `metadata.txt` = необхідний для QGIS >= 1.8.0. Містить загальну інформацію про плаґін: його версію, назву та інші метадані, необхідні для нового репозиторію плаґінів. Починаючи з QGIS 2.0 метадані з `__init__.py` не використовуються, файл `metadata.txt` є обов'язковим.

Тут знаходиться онлайновий генератор базових файлів (основи) типового плаґіна для QGIS.

Також можна скористатися плаґіном Plugin Builder, який дозволяє створювати шаблон плаґінів прямо з QGIS та не потребує доступу до інтернету. Рекомендуємо використовувати саме цей варіант, оскільки він генерує файли, сумісні з QGIS 2.0.

Попередження: Якщо ви плануєте опублікувати свій плаґін у *Офіційний репозиторій плаґінів*, переконайтесь, що плаґін відповідає деяким додатковим вимогам, необхідним для *Перевірка*

12.2 Файли плаґіна

Тут ви знайдете інформацію та приклади вмісти кожного необхідного файлу.

12.2.1 Метадані плаґіна

Перш за все Менеджер плаґінів повинен отримати основну інформацію про плаґін: назву, опис тощо. Ці дані повинні знаходитися у файлі `metadata.txt`.

Важливо: Метадані повинні записуватися з використанням UTF-8.

Назва метаданих	Необхідне	Примітки
name	True	коротка назва плаґіна
qgisMinimumVersion	True	мінімально необхідна версія QGIS у точковій нотації
qgisMaximumVersion	False	максимально підтримувана версія QGIS у точковій нотації
description	True	опис плаґіна, використання HTML не дозволяється
about	False	розширений опис плаґіна, використання HTML не дозволяється
version	True	версія плаґіна у точковій нотації
author	True	автор плаґіна
email	True	електронна пошта автора. *Не* публікується на сайті
changelog	False	може складатися з декількох рядків. Використання HTML не дозволяється
experimental	False	прапорець, `True` або `False`
deprecated	False	прапорець, `True` або `False`. Впливає на плаґін в цілому, а не на певну версію
tags	False	розділений комами список, допускаються пробіли в середині окремих теґів
homepage	False	правильний URL, що вказує на домашню сторінку плаґіна
repository	False	правильний URL, що вказує на репозиторій з вихідним кодом плаґіна
tracker	False	правильний URL, що вказує на баґтрекер плаґіна
icon	False	ім'я файлу або відносний шлях (відносно базового каталогу плаґіна)
category	False	допустимі значення *Raster*, *Vector*, *Database* та *Web*

За замовчанням плаґіни додаються у меню *Плаґіни* (нижче показується як додати плаґін до меню), але також можна розмістити плаґін у меню *Растр*, *Вектор*, *База даних* та *Інтернет*.

Відповідне значення метаданих `category` допомогає класифікувати плаґіни. Ця величина використовується як підказка для користувачів та показує де (у якому меню) шукати плаґін. Допустимі значення *Raster*, *Vector*, *Database* та *Web*. Наприклад, ваш плаґін буде доступний з меню *Растр*, тоді у файлі `metadata.txt` необхідно вказати

```
category=Raster
```

Примітка: Якщо *qgisMaximumVersion* не задано, то під час публікації в *Офіційний репозиторій плаґінів* автоматично буде використане значення рівне поточної основної версії плюс *.99*.

Приклад `metadata.txt`

```
; the next section is mandatory

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

; start of optional metadata
category=Raster
```

```
changelog=The changelog lists the plugin versions
    and their changes as in the example below:
    1.0 - First stable release
    0.9 - All features implemented
    0.8 - First testing release

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

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

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

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

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

12.2.2 __init__.py

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

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

## any other initialisation needed
```

12.2.3 mainPlugin.py

Тут відбувається вся магія, і ось як це виглядає

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

# initialize Qt resources from file resources.py
import resources

class TestPlugin:

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

  def initGui(self):
    # create action that will start plugin configuration
    self.action = QAction(QIcon(":/plugins/testplug/icon.png"), "Test plugin", self.iface.mainWindow())
```

```
    self.action.setObjectName("testAction")
    self.action.setWhatsThis("Configuration for test plugin")
    self.action.setStatusTip("This is status tip")
    QObject.connect(self.action, SIGNAL("triggered()"), self.run)

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

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

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

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

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

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

У коді плаґіна(наприклад, у `mainPlugin.py`) але обов'язково повинні буди два методи:

- `__init__` –> which gives access to Quantum GIS' interface
- `initGui()` — викликається під час активації плаґіна
- `unload()` — викликається коли плаґін деактивується

Ви можете бачити, що у нашому прикладі використовується метод **‘‘:func:‘addPluginToMenu‘<http://qgis.org/api/classQgisInterface.html#ad1af604ed4736be2bf537df58d1** Він відповідає за створення вкладеного меню плаґіна в основному меню *Модулі*. Список цих методів:

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

Їх синтаксис співпадає з синтаксисом методу `addPluginToMenu()`.

Бажано розміщувати плаґін в одному з цих меню, щоб підтримувати однорідність розміщення плаґінів. Але можна створити власний елемент головного меню, як показано нижче:

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

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

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

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

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

12.2.4 Файл ресурсів

Як ви бачили, в коді метода `initGui()` ми використовували іконку з файлу ресурсів (в нашому випадку `resources.qrc`)

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

Щоб запобігти конфліктам з іншими плаґінами або з QGIS варто вказувати префікс. Якщо цього не зробити можна отримати не той ресурс, який потрібен. Тепер можна згенерувати опис ресурсів у форматі Python. Це робиться за допомогою команди **pyrcc4**

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

Ось і все... нічого особливого :-).

Якщо ви все зробили правильно, то можете побачити свій плаґін у Менеджері плаґінів, активувати його та спостерігати повідомлення у консолі після натискання на кнопку або після вибору відповідного пункту меню.

Під час розробки реальних плаґінів краще працювати в окремому (робочому) каталозі та створити makefile, який буде генерувати інтерфейс та ресурси і копіювати плаґін до каталогу плаґінів QGIS.

12.3 Документація

Документацію до плаґінів можна писати в форматі HTML. У модулі `qgis.utils` є функція `showPluginHelp()`, яка відкриває вікно перегляду довідки, аналогічне до того, що використовується в QGIS.

Функція `showPluginHelp()` шукає файли документації у тому ж каталозі, де знаходиться модуль, що викликав її. Вона шукає файли у такому порядку `index-ll_cc.html`, `index-ll.html`, `index-en.html`, `index-en_us.html` та `index.html`, і відображає перший знайдений. тут `ll_cc` поточна локаль QGIS. Завдяки цьому плаґін може мати локалізовану документацію.

Також `showPluginHelp()` може приймати додаткові параметри: `packageName` — задає назву плаґіна, документацію якого необхідно показати, `filename` — ім'я файлу, який слід шукати замість `index` та `section` — назву якоря HTML, на який необхідно перейти після відкриття документу.

Розділ 13

Настройка IDE для розробки плаґінів

Хоча кожен програміст має улюблену IDE або текстовий редактор, ось деякі рекомендації з настройки популярних IDE для розробки плаґінів QGIS на Python.

13.1 Про настройку IDE у Windows

В Linux для розробки плаґінів додаткових настройок не потрібно. Але якщо ви працюєте в Windows, то необхідно переконатися що QGIS та інтерпретатор використовують однакові змінні оточення та бібліотеки. Найпростіший та найшвидший спосіб зробити це — відредагувати файл запуску QGIS.

Якщо використовується інсталятор OSGeo4W, файл можна знайти у каталозі `bin` директорії, де встановлено OSGeo4W. Шукайте щось на кшалт `C:\OSGeo4W\bin\qgis-unstable.bat`.

Далі описується настройка PyScripter IDE:

- зробіть копію `qgis-unstable.bat` та перейменуйте його в `pyscripter.bat`
- відкрийте цей файл у текстовому редакторі та видаліть останній рядок, який відповідає за запуск QGIS
- додайте рядок запуску PyScripter з параметром, що вказує версію Python, яку слід використовувати (у випадку QGIS 2.0 це 2.7)
- додайте ще один параметр, що вказує на каталог, де PyScripter повинен шукати бібліотеки Python QGIS. Зазвичай це каталог `bin` директорії, де встановлено OSGeo4W

```
@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
```

Тепер після запуску цього командного файлу буде встановлено необхідні змінні оточення та запущено PyScripter.

Більш популярна ніж PyScripter, Eclipse використовується багатьма розробниками. У наступному розділі ми розглянемо її настройку для розробки та тестування плаґінів. Але спочатку необхідно підготувати ваше робоче оточення. Для використання Eclipse у Windows також необхідно створити командний файл для запуску IDE.

Для цього:

- знайдіть каталог, в якому розміщено `qgis_core.dll`. Зазвичай це `C:OSGeo4Wappsqgisbin`, але якщо ви збирали QGIS самостійно, то швидше за все файл буде у каталозі `output/bin/RelWithDebInfo`
- знайдіть каталог, де знаходиться `eclipse.exe`

- створіть командний файл з наступним вмістом та використовуйте його для запуску Eclipse

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

13.2 Зневадження в Eclipse та PyDev

13.2.1 Встановлення

Для роботи з Eclipse переконайтеся, що встановили наступні програми:

- Eclipse
- Aptana Eclipse Plugin або PyDev
- QGIS 2.0

13.2.2 Підготовка QGIS

Спочатку слід підготувати QGIS. Нам знадобляться плаґіни *Remote Debug* та *Plugin reloader*.

- Зайдіть у меню *Плаґіни* → *Управління плаґінами*
- знайдіть плаґін Remote Debug (на момент написання він мав статус експериментального, то ж якщо не можете його знайти — включіть відображення експериментальних плаґінів на вкладці *Настройки*). Встановіть його.
- знайдіть плаґін Plugin Reloader і також встановіть його. Це дозволить вам перезавантажувати модуль, а не закривати та знову запускати QGIS.

13.2.3 Настройка Eclipse

Створіть новий проект у Eclipse. Ви можете вибрати *General Project* і додати реальні файли пізніше, тобто зараз немає значення де саме буде розміщено проект.

Клацніть правою клавішею миші на новому проекті та виберіть *New* → *Folder*.

Натисніть кнопку **[Advanced]** та виберіть *Link to alternate location (Linked Folder)*. Якщо ви вже маєте код, який необхідно зневаджувати, виберіть цей пункт, якщо ж не — створіть новий каталог як було написано вище.

Після цього у *Project Explorer* з'явиться дерево вихідних кодів і ви можете починати працювати з кодом. У вашому розпорядженні підсвітка синтаксису та інші інструменти IDE.

13.2.4 Настройка зневаджувача

Щоб зневаджувач запрацював відкрийте *Debug perspective* в Eclipse (*Window* → *Open Perspective* → *Other* → *Debug*).

Потім запустить зневаджувальний сервер PyDev, вибравши *PyDev* → *Start Debug Server*.

Тепер Eclipse очікує підключення до зневаджувального сервера зі сторони QGIS. Тільки-но QGIS підключиться до зневаджувального сервера, Eclipse зможе контролювати сценарії Python. Саме для цього ми встановили плаґін Remote Debug. Отже запускаємо QGIS, якщо цього зе не зроблено, а потім запускаємо модуль Remote Debug.

Після цього можна встановлювати точку зупинки, і як тільки виконання дійде до неї, ви зможете перевірити поточний стан свого плаґіна. (Точка зупинки це зелена точки на зображені нижче,

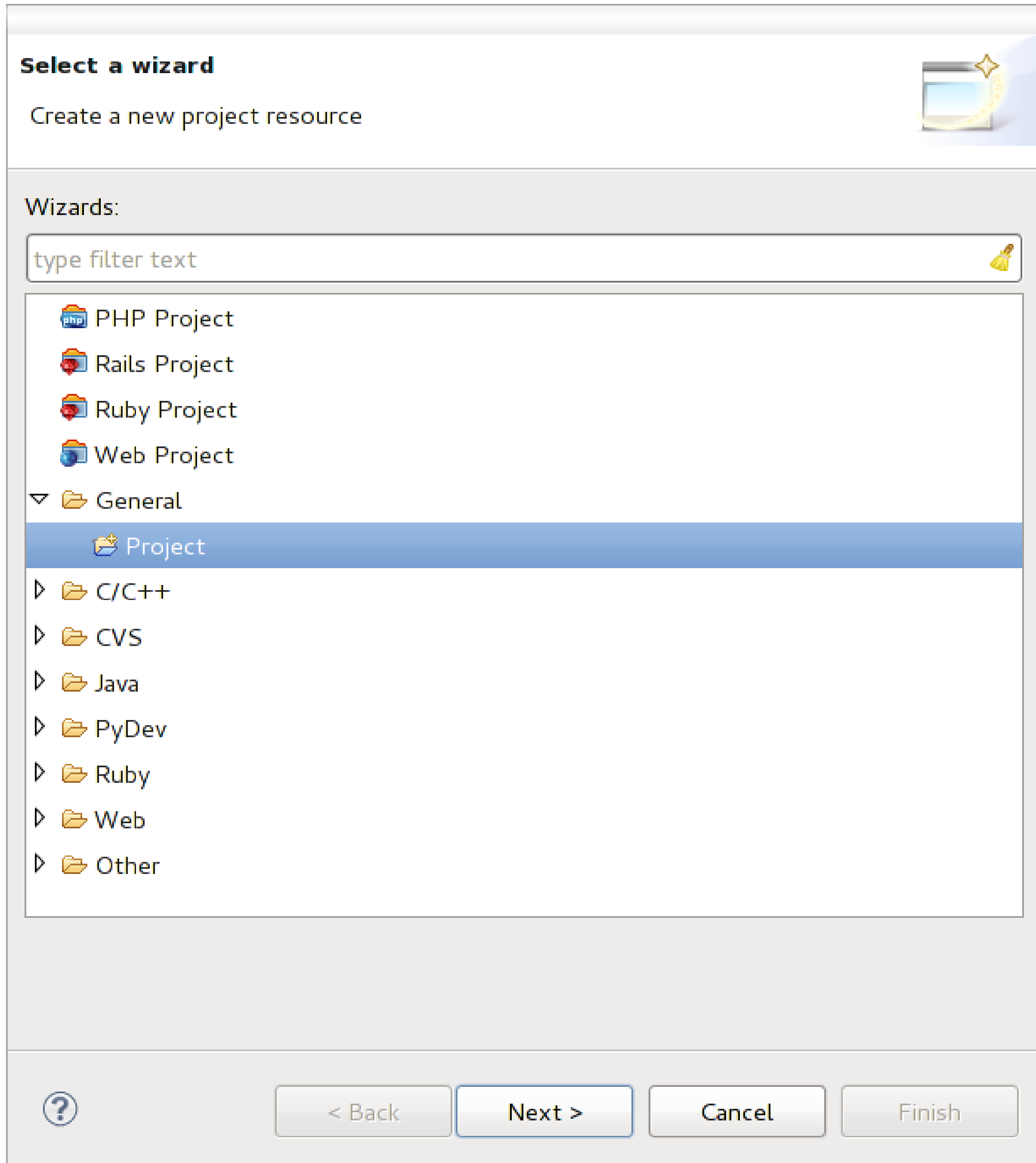

Рис. 13.1: Проект Eclipse

встановлюються вони подвійним клацанням на білій смузі поряд з рядком на якому її необхідно встановити)

```
 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 )
```

Рис. 13.2: Точка зупинки

Дуже корисним інструментом, який зараз можна використовувати, є зневаджувальна консоль. Перш ніж використовувати її переконайтесь, що виконання зараз зупинилось у точці зупинки.

Відкрийте консоль (*Window* → *Show view*). Ви побачите консоль зневаджувального сервера, де нічого цікавого немає. Але після натискання на кнопку **[Open Console]** вона перетворюється на значно цікавішу консоль зневадження PyDev. Натисніть на стрілку поруч з кнопкою **[Open Console]** та виберіть *PyDev Console*. З'явиться вікно з питанням, яку консоль ви хочете активувати. Виберіть *PyDev Debug Console*. Якщо цей пункт недоступний та вас просять запустити зневаджувач та вказати правильний фрейм, переконайтесь, що ви знаходитесь у режимі зневадження та виконання зупинилось у казаній точці зупинки.

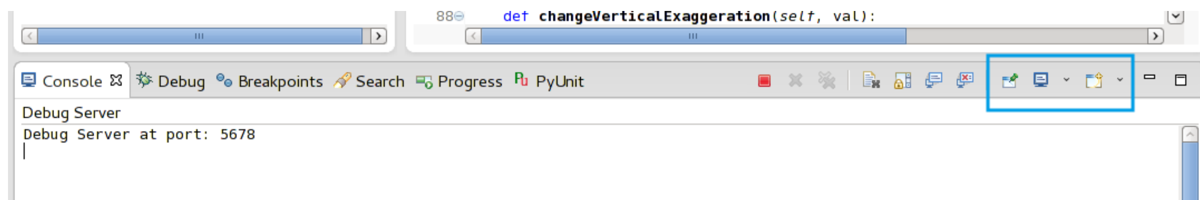

Рис. 13.3: Зневаджувальна консоль PyDev

Тепер у вас є інтерактивна консоль, у якій можна виконувати будь-які команди у поточному контексті. Ви можете маніпулювати змінними, викликати різні методи API і таке інше.

Трохи дратує те, що після кожної виконаної команди консоль переключається до зневаджувального сервера. Щоб змінити цю поведінку натисніть кнопку *Pin Console* коли знаходитись на сторінці зневаджувального сервера. Ваш вибір повинен зберегтися принаймні для поточної сесії зневадження.

13.2.5 Підключення файлів API до Eclipse

Дуже зручно, коли Eclipse знає API QGIS. Це значно зменшує кількість друкарських помилок, а крім того, дозволяє використовувати автозавершення коду викликів API.

Для цього Eclipse необхідно проаналізувати файли бібліотек QGIS та витягти з них необхідну інформацію. Вам треба тільки вказати Eclipse де саме шукати ці бібліотеки.

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

У верхній частині вікна знаходяться налаштовані інтерпретатори Python (у нашому випадку тільки Python 2.7 для QGIS), а в нижній частині — декілька вкладок. Нас цікавлять вкладки *Libraries* та *Forced Builtins*.

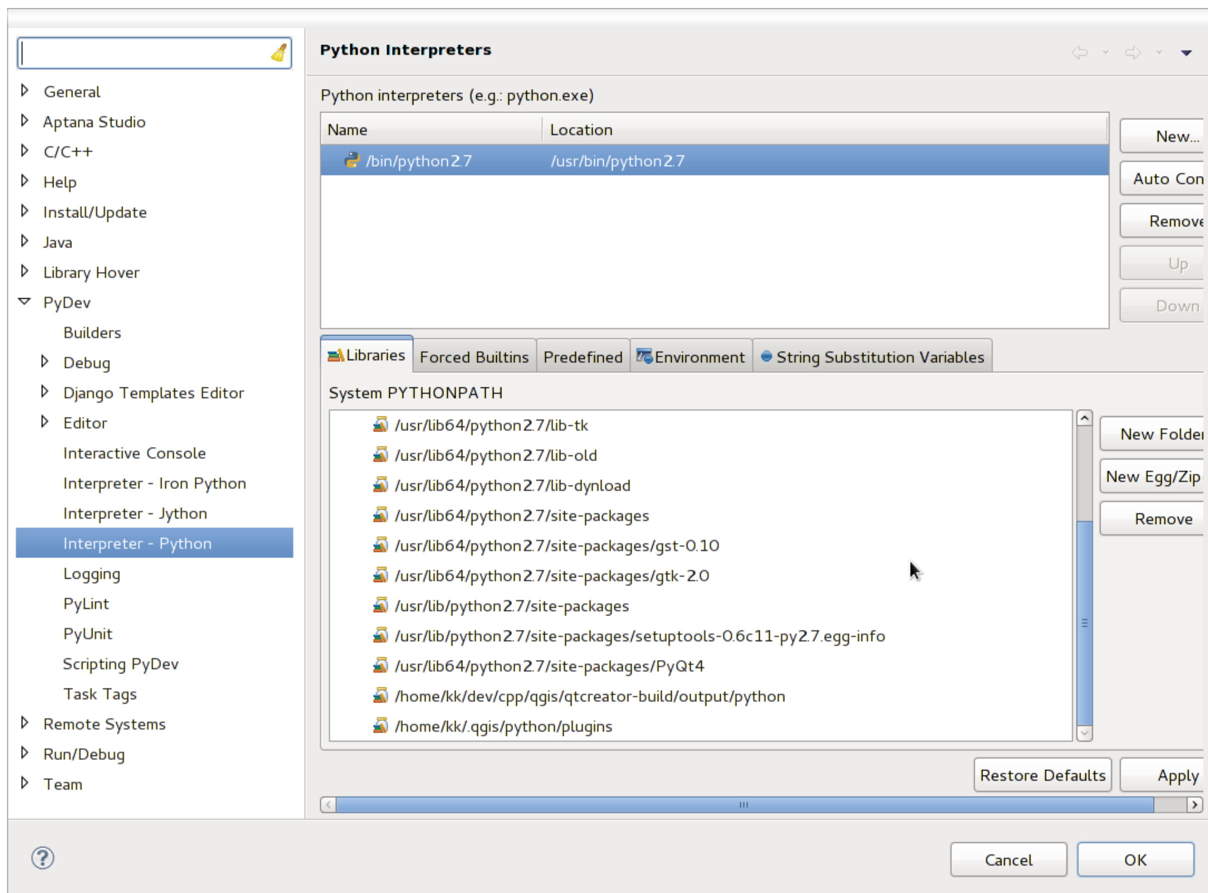

Рис. 13.4: Зневаджувальна консоль PyDev

Спочатку перейдіть на вкладку *Libraries*. Натисніть **[New Folder]** та вкажіть каталог Python вашої встановленої QGIS. Якщо ви не знаєте де цей каталог знаходиться (це не каталог плаґінів!), відкрийте QGIS, активуйте консоль Python та виконайте команду `qgis`. Результатом цієї команди буде список плаґінів та їх каталоги. Відкиньте `/qgis/__init__.pyc` з цього шляху та отримаєте шуканий каталог.

Також необхідно додати каталог плаґінів (в Linux це `~/.qgis2/python/plugins`).

Тепер перейдіть на вкладку *Forced Builtins*, натисніть **[New...]** та введіть `qgis`. Це змусить Eclipse проаналізувати API QGIS. Швидше за все ви також захочете щоб Eclipse знала й API PyQt4. Тому додайте PyQt4 як *forced builtin*. Можливо, цей пункт уже буде присутній на вашій вкладці.

Натисніть **[OK]** щоб закінчити.

Зверніть увагу: кожного разу, коли API QGIS змінюється (наприклад, коли ви перезібрали QGIS і файли SIP змінилися), необхідно повертатися на ці вкладки та натискати **[Apply]** щоб Eclipse проаналізували бібліотеки знову.

Інший варіант настройки Eclipse для роботи з плаґінами QGIS описано тут.

13.3 Зневадження з PDB

Якщо ви не користуєтесь такими IDE як Eclipse, ви можете зневаджувати за допомогою PDB. Для цього:

додайте наступний код, у місце, яке необхідно зневадити

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

тепер запустить QGIS з командного рядка

в Linux виконайте:

$ ./Qgis

в Mac OS X виконайте:

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

Як тільки програма дійде до вашої точки зупинки, консоль стане доступною і ви зможете виконувати будь-які команди!

TODO: Add testing information

Розділ 14

Шари плаґінів

Якщо плаґін використовує свої власні методи рендерінгу шарів карти, зручніше за все створити свій власний тип шарів на основі `QgsPluginLayer`.

TODO: Check correctness and elaborate on good use cases for QgsPluginLayer, ...

14.1 Успадкування `QgsPluginLayer`

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

```
class WatermarkPluginLayer(QgsPluginLayer):

  LAYER_TYPE="watermark"

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

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

За необхідності можна додати методи для запису та зчитування інформації з файлу проекту

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

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

Для завантаження проекту, що містить такий шар, необхідна наявність спеціального класу

```
class WatermarkPluginLayerType(QgsPluginLayerType):

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

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

Також можна додати код для відображення додаткової інформації у вікні параметрів шару

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

Розділ 15

Сумісність з попередніми версіями QGIS

15.1 Меню плаґіна

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

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

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

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

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

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

Розділ 16

Публікація плаґіна

Якщо після створення плаґіна ви вирішите, що він може знадобитися й іншим користувачам, не бійтеся опублікувати його в *Офіційний репозиторій плаґінів*. На цій сторінці ви також знайдете інструкції з підготовки пакету, слідуванням яким позбавить вас від проблем з встановленням плаґіна через Менеджер плаґінів. Якщо ж вам необхідно створити свій власний репозиторій — створіть простий файл XML, який описує всі плаґіни та їх метадані. Приклад такого файлу можна знайти на сторінці Python Plugin Repositories.

16.1 Офіційний репозиторій плаґінів

Офіційний репозиторій плаґінів знаходиться за адресою http://plugins.qgis.org/.

Щоб повноцінно користуватися офіційним репозиторієм, необхідно отримати OSGeo ID на порталі OSGeo.

Після завантаження плагіна на сервер, він буде перевірений та схвалений адміністрацією і ви отримаєте повідомлення.

TODO: Insert a link to the governance document

16.1.1 Права

Офіційний репозиторій плаґінів працює за такими правилами:

- кожний зареєстрований користувач може додавати плагіни
- *адміністратори* можуть схвалювати та відкликати всі версії плаґінів
- всі версії плаґінів користувачів, які мають дозвіл *plugins.can_approve*, схвалюються автоматично після завантаження на сервер
- користувачі, що мають дозвіл *plugins.can_approve* і знаходяться у списку *власників* плаґіна, можуть схвалювати версії цього плаґіна, завантажені іншими користувачами
- плаґін може бути видалений з репозиторію лише *адміністрацією* та *власником* плаґіна
- якщо користувач без дозволу *plugins.can_approve* завантажує нову версію плаґіна, то ця версія не буде схвалена, навіть якщо плаґін був схвалений раніше

16.1.2 Довірче управлення

Адміністрація може *довіряти* окремим авторам плаґінів шляхом надання дозволу *plugins.can_approve*.

Панель адміністрування дозволяє видати необхідні дозволи як автору так і всім *власникам* плаґіна.

16.1.3 Перевірка

Під час завантаження плаґінів на сервер їх метадані автоматично імпортуються та перевіряться.

Ось список основних правил перевірки, про які необхідно знати перед публікацією плаґіна в офіційному репозиторії:

1. the name of the main folder containing your plugin must contain only contains ASCII characters (A-Z and a-z), digits and the characters underscore (_) and minus (-), also it cannot start with a digit
2. обов'язково повинен бути файл `metadata.txt`
3. всі обов'язкові метадані, описані в *metadata table*, повинні бути заповненими
4. значення поля *version* метаданих повинно бути унікальним

16.1.4 Структура плаґіна

Щоб пройти перевірку, стиснений (`.zip`) пакет плагіна повинен мати певну структуру. Оскільки плаґіни розпаковуються у домашньому каталозі користувача, до директорії плаґінів, вони повинні знаходитися кожний у своєму каталозі всередині файлу `.zip`. Обов'язковими файлами є лише `metadata.txt` та `__init__.py`. Але не завадить також мати `README` та іконку. Нижче показано структуру пакета плаґіна

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

Розділ 17

Фрагменти коду

Тут зібрано фрагменти коду, які будуть корисні під час розробки плаґінів.

17.1 Як викликати метод за комбінацією клавіш

Додайте в `initGui()`

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

Та до `unload()`

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

Метод, що викликається після натискання F7

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

17.2 Як керувати видимістю шарів

Починаючи з QGIS 2.4 доступне нове API, яке дозволяє отримати доступ до елементів легенди. Нижче наведено приклад перемикання видимості поточного шару

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

17.3 Як отримати доступ до таблиці атрибутів вибраних об'єктів

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

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

Метод приймає один параметр (нове значення атрибута) та викликається так

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

Розділ 18

Бібліотека аналізу мереж

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

- дозволяє створювати математичний граф з географічних даних (лінійних векторних шарів)
- реалізує базові методи теорії графів (на сьогодні лише алгоритм Дейкстри)

Бібліотека аналізу мереж з'явилась як результат експорту основних функцій плаґіна RoadGraph і тепер ви можете використовувати його можливості у своїх модулях або з консолі Python.

18.1 Загальна інформація

Типовий алгоритм використання бібліотеки складається з наступних кроків:

1. створити граф з просторових даних (зазвичай лінійних векторних шарів)
2. проаналізувати граф
3. використати результати аналізу (наприклад. візуалізувати їх)

18.2 Побудова графу

Перш за все необхідно підготувати вхідні дані, тобто конвертувати векторний шар у граф. Всі подальші операції будуть виконуватися з цим графом, а не з шаром.

В якості джерела даних може виступати будь-який векторний шар. Вузли ліній стануть вузлами графу, а сегменти ліній — ребрами. Якщо декілька вершин мають однакові координати, то вони будуть відповідати одній вершині графу. Тобто дві лінії, що мають спільний вузол, будуть зв'язані між собою.

Крім того, під час створення графу можна «прив'язати» до векторного шару будь-яку кількість додаткових точок. Для кожної такої додаткової точки буде знайдено відповідність — найближчий вузол чи ребро графу. В останньому випадку ребро буде розбито на дві частини, і з'явиться новий вузол.

В якості характеристик ребер можуть використовуватися атрибути векторного шару або довжина ребра.

Конвертор з векторного шару у граф реалізовано з використанням шаблону програмування будівельник. А за побудову графу відповідає так званий «директор». На сьогодні доступний лише один директор QgsLineVectorLayerDirector. Директор задає основні налаштування, які будуть використовуватися під час конвертації шару в граф, та за допомогою будівельника будує граф. Як і у випадку з директором, на сьогодні доступний лише один будівельник QgsGraphBuilder, який генерує об'єкти QgsGraph. Ви можете реалізувати своїх власних будівельників, які будуть генерувати графи, сумісні з такими бібліотеками як BGL та NetworkX.

Для обчислення характеристик ребер використовується шаблон проектування стратегія. На сьогодні доступна лише стратегія QgsDistanceArcProperter, яка враховує довжину маршруту. За необхідності ви можете реалізувати власну стратегію, яка буде використовувати все необхідні параметри. Наприклад, плаґін RoadGraph використовує стратегію, що обчислює час подорожі на основі довжини ребер та швидкості з атрибутів шару.

Розглянемо процес створення графу докладніше.

Спочатку необхідно імпортувати модуль аналізу мереж

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

Та повідомити директору про неї. Один директор може використовувати декілька стратегій

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

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

У конструктор директора передається векторний шар, на основі якого буде створено граф, а також інформація про характер руху на кожному сегменті дороги (дозволені напрямки руху, односторонній чи двосторонній рух).

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

Розглянемо ці параметри:

- `vl` — векторний шар, на основі якого створюється граф
- `directionFieldId` — індекс поля таблиці атрибутів, у якому знаходиться інформація про напрямки руху. Якщо вказано `-1`, ця інформація не використовується
- `directDirectionValue` — значення поля, яке відповідає прямому напрямку руху (рух від першої вершини до останньої)
- `directDirectionValue` — значення поля, яке відповідає прямому напрямку руху (рух від першої вершини до останньої)
- `bothDirectionValue` —значення поля, яке відповідає двосторонньому руху (тобто допускається як рух від першої вершини до останньої, так і рух в зворотньому напрямі, від останньої вершини до першої)
- `defaultDirection` — напрям руху за замовчанням. Ця величина використовується для тих ділянок доріг, для яких значення поля `directionFieldId` не задане або не співпадає з жодним з вищенаведених.

Далі необхідно створити стратегію обчислення характеристик ребер графу

```
properter = QgsDistanceArcProperter()
```

Та повідомити директору про неї. Один директор може використовувати декілька стратегій

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

Нарешті створюємо будівельника, який буде будувати граф. Конструктор `QgsGraphBuilder` приймає наступні параметри:

- `crs` — система координат, що буде використовуватися. Обов'язковий параметр.
- `otfEnabled` — вказує на необхідність використання перепроектування «на льоту». За замовчанням `True` (використовувати перепроектування).
- `topologyTolerance` — топологічний допуск. За замовчанням 0.
- `ellipsoidID` — еліпсоїд, який буде використовуватися. За замовчанням WGS 84.

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

Також можна задати одну або декілька точок, які будуть використовуватися під час аналізу. Наприклад

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

Тепер будуємо граф та «прив'язуємо» до нього точки

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

Побудова графу може зайняти деякий час (залежить від кількості об'єктів у шарі та розмірів власне шару). До `tiedPoints` записуються координати «прив'язаних» точок. Коли операція завершиться ми отримаємо граф придатний для подальшого аналізу

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

Тепер можна отримати індекси наших точок

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

18.3 Аналіз графу

В основі аналізу мереж лежить задача зв'язності графу та задача пошуку найкоротшого маршруту. Для вирішення цих задач у бібліотеці аналізу мереж реалізовано алгоритм Дейкстри.

Алгоритм Дейкстри знаходить найкоротший маршрут від однієї вершини графу до всіх інших а також значення параметру, що оптимізується. Наочно результат можна представити як дерево найкоротших маршрутів.

Дерево найкоротших маршрутів — це орієнтований зважений граф (або більш точно — дерево) з наступними властивостями:

- тільки одна вершина не має вхідних ребер — корінь дерева
- всі інші вершини мають лише одне вхідне ребро
- якщо до вершини B можна дістатися з вершини A, то шлях, який їх з'єднує, єдиний і він же найкоротший (оптимальний) на вихідному графі

Дерево найкоротших шляхів можна отримати за допомогою методів `shortestTree()` та `dijkstra()` класу QgsGraphAnalyzer. Рекомендується використовувати метод `dijkstra()`, оскільки він працює швидше та ефективніше використовує пам'ять.

Метод `shortestTree()` може бути корисний, коли необхідно обійти дерево найкоротших маршрутів. Він створює новий об'єкт (завжди `QgsGraph`) та приймає три параметри:

- `source` — вихідний граф
- `startVertexIdx` — індекс точки на графі (корінь дерева)
- `criterionNum` — порядковий номер характеристики ребра (відлік починається з 0).

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

Метод `dijkstra()` має такі ж параметри, але повертає два масиви. У першому масиві `i`-тий елемент містить індекс ребра, що входить в `i`-ту вершину, в протилежному випадку — `-1`. У другому масиві `i`-тий елемент містить відстань від `i`-ї вершини, якщо до вершини можна дістатися з кореня, або максимально велике значення, яке може вміститися у тип C++ double, якщо вершина недосяжна.

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

Ось дуже простий спосіб відобразити дерево найкоротших маршрутів з використанням графу, отриманого в результаті роботи метода `shortestTree()` (не забудьте замінити координати початкової точки на свої, а також виберіть шар доріг у легенді). **Увага**: використовуйте цей код тільки як приклад і лише для невеликих шарів, так як він генерує велику кількість об'єктів QgsRubberBand

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

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

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

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

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

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

Same thing but using `dijkstra()` method:

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

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

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

pStart = QgsPoint(-1.37144, 0.543836)
```

```
tiedPoint = director.makeGraph(builder, [pStart])
pStart = tiedPoint[0]

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

for edgeId in tree:
  if edgeId == -1:
    continue
  rb = QgsRubberBand(qgis.utils.iface.mapCanvas())
  rb.setColor (Qt.red)
  rb.addPoint (graph.vertex(graph.arc(edgeId).inVertex()).point())
  rb.addPoint (graph.vertex(graph.arc(edgeId).outVertex()).point())
```

18.3.1 Пошук найкоротшого маршруту

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

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

На цьому пошук маршруту завершено. Ми отримали інвертований список вершин (тобто вершини йдуть у зворотньому порядку, від кінцевої точки до початкової), які будуть відвідані під час руху по цьому маршруту.

Ось приклад пошуку найкоротшого маршруту для консолі Python QGIS (не забудьте замінити координати початкової та кінцевої точок на свої значення. та виберіть шар доріг у легенді) з використанням методу `shortestTree()`

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

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

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

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

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

tStart = tiedPoints[0]
```

```
tStop = tiedPoints[1]

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

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

if idStop == -1:
  print "Path not found"
else:
  p = []
  while (idStart != idStop):
    l = tree.vertex(idStop).inArc()
    if len(l) == 0:
      break
    e = tree.arc(l[0])
    p.insert(0, tree.vertex(e.inVertex()).point())
    idStop = e.outVertex()

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

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

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

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

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

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

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

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

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

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

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

if tree[idStop] == -1:
  print "Path not found"
else:
  p = []
  curPos = idStop
  while curPos != idStart:
```

```
    p.append(graph.vertex(graph.arc(tree[curPos]).inVertex()).point())
    curPos = graph.arc(tree[curPos]).outVertex();

  p.append(tStart)

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

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

18.3.2 Пошук областей доступності

Назвемо областю доступності вершини графу A таку підмножину вершин графу, до яких можна дістатися з вершини A та вартість оптимального шляху від A до елементів цієї множини не буде перевищувати певної наперед заданої величини.

Більш наочно це визначення можна пояснити наступним прикладом. Є пожежне депо. У яку частини міста зможе попасти пожежне авто за 5 хвилин, 10 хвилин, 15 хвилин? Відповіддю на ці питання і буде область досяжності пожежного депо.

Пошук областей досяжності легко організувати за допомогою методу `dijksta()` класу `QgsGraphAnalyzer`. Достатньо порівняти елементи масиву вартості з необхідною величиною. Якщо `cost[i]` менше заданої величини або дорівнює їй, то `i`-та вершина графу входить у множину доступності, в протилежному випадку — не входить.

Не настільки очевидним є пошук меж області доступності. Нижня межа доступності — множина вершин, до яких ще можна дістатися, а верхня межа — множина недосяжних вершин. Насправді все дуже просто: межа доступності проходить по таким ребрам графу, для яких вхідна вершина ще доступна, а вихідна — ні.

Ось приклад

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

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

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

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

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

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

idStart = graph.findVertex(tStart)
```

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

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

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

автономні програми
- запуск, 3
- Python, 2

друк карти, 37
файли GPX
- завантаження, 6

фільтрація, 42
геометрія
- доступ, 27
- обробка, 26
- предикати та операції, 28
- створення, 27

геометрії MySQL
- завантаження, 6

градуйований знак, 20
карта, 32
- архітектура, 33
- гумові полоси, 35
- інструменти карти, 34
- маркери вершин, 35
- створення власних елементів карти, 37
- створення власних інструментів карти, 36
- вбудовування, 33

консоль
- Python, 1

метадані, 56
настройки
- читання, 45
- глобальні, 47
- проект, 47
- шар, 48
- збереження, 45

об'єкти
- векторні шари перегляд, 13

обчислення значень, 42
оновлення
- растрові шари, 11

перегляд
- об'єкти, векторні шари, 13

плаґіни, 69
- документація, 58
- доступ до атрибутів вибраних об'єктів, 71
- файл ресурсів, 58
- фрагменти коду, 58
- написання, 53
- написання коду, 54
- офіційний репозиторій плаґінів, 69
- реалізація довідки, 58
- реліз, 64
- розробка, 51
- тестування, 64
- видимість шарів, 71
- виклик метода за комбінацією клавіш, 71
- metadata.txt, 54, 56

проекції, 32
просторовий індекс
- використання, 16

растр WMS
- завантаження, 6

растри
- багатоканальні, 11
- одноканальні, 10

растрові шари
- інформація, 9
- оновлення, 11
- стиль відображення, 9
- використання, 7
- запит, 11
- завантаження, 6

реєстр шарів карти, 7
- додавання шарів, 7

рендерери
- власні, 24

рендерінг карти, 37
- простий, 39

символ
- робота з, 21

символьні шари
- робота з, 21
- власні типи, 22

системи координат, 31
стиль
- градуйований знак, 20
- стара, 26
- унікальний знак, 20
- single symbol renderer, 19

шари OGR
- завантаження, 5

шари PostGIS
- завантаження, 5

шари SpatiaLite
    завантаження, 6
шари плаґінів, 64
    успадковування QgsPluginLayer , 65
шари з тексту з роздільниками
    завантаження, 5
унікальний знак, 20
векторні шари
    перегляд об'єкти, 13
    редагування, 14
    стиль, 19
    завантаження, 5
    збереження, 17
вирази, 42
    аналіз, 43
    обчислення, 44
вивід
    макети карти, 39
    растрове зображення, 41
    PDF, 41
власні
    рендерери, 24
запит
    растрові шари, 11
запуск
    автономні програми, 3
завантаження
    файли GPX, 6
    геометрії MySQL, 6
    растр WMS, 6
    растрові шари, 6
    шари OGR, 5
    шари PostGIS, 5
    шари SpatiaLite, 6
    шари з тексту з роздільниками, 5
    векторні шари, 5

API, 1

memory провайдер, 18
metadata.txt, 56

Python
    автономні програми, 2
    консоль, 1
    плагіни, 1
    розробка плаґінів, 51

resources.qrc, 58

single symbol renderer, 19