PyQGIS developer cookbook

Выпуск 2.8

QGIS Project

24 October 2015

Глава 1

Введение

Этот документ задумывался как учебник и справочное пособие. Вы не найдёте здесь описания всех возможных вариантов использования, это скорее обзор основных функциональных возможностей.

Начиная с версии 0.9, в QGIS появилась возможность поддержки сценариев на языке программирования Python. Мы выбрали Python, так как это один из наиболее известных скриптовых языков. Привязки (bindings) PyQGIS зависят от SIP и PyQt4. Основная причина использования SIP вместо более распространенного SWIG состоит в том, что код QGIS зависит от библиотек Qt. Привязки Python к Qt (PyQt) также создаются с использованием SIP, что позволяет обеспечить прозрачную интерграцию PyQGIS и PyQt.

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

Есть несколько способов программирования на Python в QGIS, подробнее они будут рассмотрены в следующих разделах:

- automatically run Python code when QGIS starts
- ввод команд в консоли Python QGIS
- создание и использование расширений на Python
- создание собственного приложения на базе QGIS API

Существует полное описание QGIS API, в котором собрана информация обо всех классах библиотек QGIS. QGIS Python API практически идентично C++ API.

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

1.1 Run Python code when QGIS starts

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

1.1.1 PYQGIS_STARTUP environment variable

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

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

1.1.2 The `startup.py` file

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

1.2 Консоль Python

Для небольших сценариев можно воспользоваться встроенной консолью Python. Открыть её можно из меню: *Модули* → *Консоль Python*. Консоль откроется как немодальное окно:

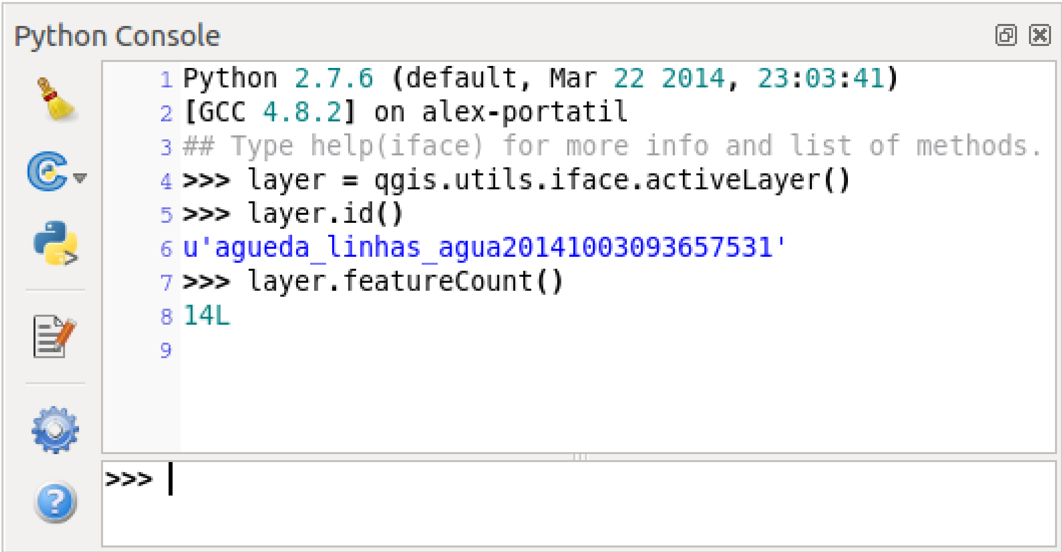

Рис. 1.1: QGIS Python console

The screenshot above illustrates how to get the layer currently selected in the layer list, show its ID and optionally, if it is a vector layer, show the feature count. For interaction with QGIS environment, there is a `iface` variable, which is an instance of `QgsInterface`. This interface allows access to the map canvas, menus, toolbars and other parts of the QGIS application.

For convenience of the user, the following statements are executed when the console is started (in future it will be possible to set further initial commands)

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

Тем, кто использует консоль часто, стоит назначить комбинацию клавиш для её вызова (в меню *Установки* → *Комбинации клавиш...*)

1.3 Расширения на Python

QGIS allows enhancement of its functionality using plugins. This was originally possible only with C++ language. With the addition of Python support to QGIS, it is also possible to use plugins written in Python. The main advantage over C++ plugins is its simplicity of distribution (no compiling for each platform needed) and easier development.

Many plugins covering various functionality have been written since the introduction of Python support. The plugin installer allows users to easily fetch, upgrade and remove Python plugins. See the Python Plugin Repositories page for various sources of plugins.

Создание расширений на Python — это просто, см. *Разработка расширений на Python*.

1.4 Приложения на Python

При обработке ГИС данных часто удобнее создать несколько сценариев, автоматизирующих процесс, чем постоянно выполнять одни и те же действия. Это более чем возможно при использовании PyQGIS — просто импортируйте модуль `qgis.core`, инициализируйте его и всё готово к обработке.

Или же вам может потребоваться интерактивное приложение, обладающее некоторым функционалом ГИС — измерение данных, экспорт карты в формат PDF или что-то ещё. Модуль `qgis.gui` предоставляет различные элементы интерфейса, наиболее важный среди них — виджет карты, который легко интегрируется в приложение и поддерживает масштабирование, панорамирование и/или любые другие инструменты для работы с картой.

1.4.1 Использование PyQGIS в приложениях

Примечание: *не* используйте имя `qgis.py` для своих сценариев — Python не сможет импортировать привязки, так как имя сценария будет "затенять" их.

Прежде всего нужно импортировать модуль qgis и задать путь, где QGIS будет искать ресурсы — базу проекций, провайдеров и др. Если при установке путей поиска второй аргумент задан как `True`, QGIS инициализирует все пути стандартными значениями с использованием заданного префикса. Вызов функции `initQgis()` очень важен, так как позволят QGIS выполнить поиск доступных провайдеров данных.

```
from qgis.core import *

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

# load providers
QgsApplication.initQgis()
```

Теперь можно работать с API QGIS — загружать слои, выполнять какую-то обработку или создать графическое приложение с картой. Возможности бесконечны :-)

После окончания работы с библиотеками QGIS вызовите `exitQgis()`, чтобы быть уверенными, что все ресурсы были освобождены (например, что список слоев карты очищен и все слои удалены):

```
QgsApplication.exitQgis()
```

1.4.2 Запуск приложений

Необходимо указать системе где искать библиотеки QGIS и соответствующие модули Python — иначе при запуске появится сообщение об ошибке:

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

Для этого необходимо установить переменную окружения `PYTHONPATH`. В приведенных ниже командах `qgispath` необходимо заменить на реальный путь к каталогу с установленной QGIS:

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

Теперь путь к модулям PyQGIS известен, но они в свою очередь зависят от библиотек `qgis_core` и `qgis_gui` (модули Python служат всего лишь "обёртками" над этими библиотеками). Обычно, операционной системе неизвестно расположение этих библиотек, поэтому вы получите ошибку импорта еще раз (сообщение может отличаться в зависимости от системы):

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

Проблема решается путем добавления каталогов с библиотеками QGIS в путь поиска линковщика:

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

Эти команды можно вписать в загрузочный скрипт, который будет настраивать систему перед запуском приложения. При развертывании приложений, использующих PyQGIS, можно использовать один из двух способов:

- требовать от пользователя перед инсталляцией вашего приложения выполнять установку QGIS. Установщик приложения должен выполнять поиск каталогов с библиотеками QGIS и позволять пользователю задать эти каталоги вручную. Преимуществом такого подхода является простота, однако, он требует от пользователя выполнения дополнительных действий.
- поставлять QGIS вместе со своим приложением. Подготовка в выпуску станет более сложной и размер приложения возрастет, но зато пользователи будут избавлены от необходимости загружать и устанавливать дополнительное программное обеспечение.

Эти два подхода можно комбинировать — можно развертывать самостоятельное приложение в Windows и Mac OS X, а в Linux оставить установку QGIS на попечении пользователя и пакетного менеджера.

Loading Projects

Sometimes you need to load an existing project from a plugin or (more often) when developing a stand-alone QGIS Python application (see: *Приложения на Python*).

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

```
# If you are not inside a QGIS console you first need to import
# qgis and PyQt4 classes you will use in this script as shown below:
from qgis.core import QgsProject
from PyQt4.QtCore import QFileInfo
# Get the project instance
project = QgsProject.instance()
# Print the current project file name (might be empty in case no projects have been loaded)
print project.fileName
u'/home/user/projects/my_qgis_project.qgs'
# Load another project
project.read(QFileInfo('/home/user/projects/my_other_qgis_project.qgs'))
print project.fileName
u'/home/user/projects/my_other_qgis_project.qgs'
```

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

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

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

Загрузка слоёв

Давайте загрузим несколько слоёв с данными. В QGIS слои делятся на векторные и растровые. Кроме того, существуют пользовательские типы слоёв, но их обсуждение выходит за рамки этой книги.

3.1 Векторные слои

Чтобы загрузить векторный слой нужно указать идентификатор источника данных имя слоя и название провайдера:

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

Идентификатор источника данных это строка, специфичная для каждого провайдера векторных данных. Имя слоя используется в виджете списка слоёв. Необходимо проверять успешно ли завершилась загрузка слоя или нет. В случае каких-либо ошибок возвращается неправильный объект.

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

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

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

Ниже показано как получить доступ к различным источникам данных используя провайдеры векторных данных:

- 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")
```

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

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

3.2 Растровые слои

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

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

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

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

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

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

Raster layers can also be created from a WCS service.

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

detailed URI settings can be found in provider documentation

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

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

3.3 Список слоёв карты

Если вы хотите использовать открытые слои при отрисовке карты — не забудьте добавить их к списку слоёв карты. Список слоёв карты станет их владельцем, а получить доступ к ним можно будет из любой части приложения по уникальному идентификатору. При удалении слоя из списка слоёв карты, происходит его уничтожение.

Adding a layer to the registry

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

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

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

For a list of loaded layers and layer ids, use

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

TODO: More about map layer registry?

Работа с растровыми слоями

Этот раздел описывает различные действия, которые можно выполнять с растровыми слоями.

4.1 Информация о слое

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

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

4.2 Стиль отображения

Сразу после загрузки растровый слой отображается стилем, основанным на его типе. Стиль отображения может быть изменён в диалоге свойств растрового слоя или программным путем. Существуют следующие стили отображения:

Индекс	Константа: QgsRasterLater.X	Комментарий
1	SingleBandGray	Одноканальное изображение отображается в оттенках серого цвета
2	SingleBandPseudoColor	Одноканальное изображение отображается с использованием псевдоцвета
3	PalettedColor	Одноканальное изображение отображается с использованием псевдоцвета
4	PalettedSingleBandGray	Слой с "палитрой", отрисовка в оттенках серого
5	PalettedSingleBandPseudoColor	"Palette" layer drawn using a pseudocolor algorithm
7	MultiBandSingleBandGray	Слой состоит из 2 и более каналов, но отображается только один канал в оттенках серого
8	MultiBandSingleBandPseudoColor	Слой состоит из 2 и более каналов, но отображается только один канал с использованием псевдоцвета
9	MultiBandColor	Слой состоит из 2 и более каналов, установлено соответствие с цветами пространства RGB

To query the current drawing style:

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

Одноканальные растровые слои могут отображаться либо в оттенках серого (малые значения = черный, большие значения = белый) или с использованием псевдоцвета, когда одинаковым значениям присваивается свой цвет. Кроме того, одноканальные растры могут отображаться с использованием палитры. При отображении многоканальных слоёв обычно устанавливается соответствие между каналами и цветами пространства RGB. Ещё один способ — использование одного канала для отрисовки в оттенках серого или в псевдоцвете.

В следующих разделах описано как узнать и изменить стиль отображения слоя. После того, как изменения внесены, потребуется обновить карту, см. *Обновление слоёв*.

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

4.2.1 Одноканальные растры

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

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

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

- linear (`INTERPOLATED`): resulting color is linearly interpolated from the color map entries above and below the actual pixel value
- discrete (`DISCRETE`): color is used from the color map entry with equal or higher value
- точный (`EXACT`): цвета не интерполируются, отображаются только пиксели со значениями, равными значениям цветовой карты

Установка градиента от зеленого к желтому цвету (для значений от 0 до 255) выглядит так:

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

To return back to default gray levels, use:

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

4.2.2 Многоканальные растры

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

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

4.3 Обновление слоёв

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()
```

Первая конструкция нужна для того, чтобы убедиться, что при использовании кеша отрисовки кешированные изображения обновляемого слоя удалены. Этот функционал доступен начиная с QGIS 1.4, в более ранних версиях такой функции нет — поэтому, в начале, чтобы быть уверенными в работоспособности кода во всех версиях QGIS, выполняется проверка на существование метода.

Вторая конструкция вызывает сигнал, который заставляет все карты, содержащие слой, выполнить перерисовку.

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

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

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

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

4.4 Получение значений

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()
```

The `results` method in this case returns a dictionary, with band indices as keys, and band values as values.

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

Работа с векторными слоями

Этот раздел описывает различные действия, которые можно выполнять с векторными слоями.

5.1 Retrieving informations about attributes

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

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

5.2 Selecting features

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

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

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

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

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

To clear the selection, just pass an empty list:

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

5.3 Обход объектов векторного слоя

Iterating over the features in a vector layer is one of the most common tasks. Below is an example of the simple basic code to perform this task and showing some information about each feature. the `layer` variable is assumed to have a `QgsVectorLayer` object

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

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

    # fetch attributes
    attrs = feature.attributes()

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

5.3.1 Accessing attributes

Attributes can be referred to by their name.

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

Alternatively, attributes can be referred to by index. This is will be a bit faster than using the name. For example, to get the first attribute:

```
print feature[0]
```

5.3.2 Iterating over selected features

if you only need selected features, you can use the `selectedFeatures()` method from vector layer:

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

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

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

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

5.3.3 Iterating over a subset of features

If you want to iterate over a given subset of features in a layer, such as those within a given area, you have to add a QgsFeatureRequest object to the getFeatures() call. Here's an example

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

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

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

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

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

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

5.4 Редактирование векторных слоёв

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

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

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

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

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

5.4.1 Добавление объектов

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

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

5.4.2 Удаление объектов

To delete some features, just provide a list of their feature IDs

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

5.4.3 Изменение объектов

It is possible to either change feature's geometry or to change some attributes. The following example first changes values of attributes with index 0 and 1, then it changes the feature's geometry

```
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 })
```

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

5.4.4 Добавление и удаление полей

To add fields (attributes), you need to specify a list of field definitions. For deletion of fields just provide a list of field indexes.

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

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()
```

5.5 Редактирование векторных слоёв с использованием буфера изменений

При редактировании векторных данных в QGIS, сначала необходимо перевести соответствущий слой в режим редактирования, затем внести изменения и, наконец, зафиксировать (или отменить) эти изменения. Все сделанные изменения не применяются до тех пор, пока вы их не зафиксируете — они хранятся в буфере изменений слоя. Данную возможность можно использовать и программно — это всего лишь другой способ редактирования векторных слоёв, дополняющий прямой доступ к провайдеру. Использовать этот функционал стоит тогда, когда пользователю предоставляются графические инструменты редактирования, чтобы он мог решить когда фиксировать/отменять изменения, а также мог использовать инструменты повтора/отмены. При фиксации изменений, все имеющиеся в буфере операции будут переданы провайдеру.

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

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

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

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

In order to make undo/redo work properly, the above mentioned calls have to be wrapped into undo commands. (If you do not care about undo/redo and want to have the changes stored immediately, then you will have easier work by *editing with data provider*.) How to use the undo functionality

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

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

if problem_occurred:
    layer.destroyEditCommand()
   return

# ... more editing ...

layer.endEditCommand()
```

The `beginEditCommand()` will create an internal "active" command and will record subsequent changes in vector layer. With the call to `endEditCommand()` the command is pushed onto the undo stack and the user will be able to undo/redo it from GUI. In case something went wrong while doing the changes, the `destroyEditCommand()` method will remove the command and rollback all changes done while this command was active.

Для актививации режима редактирования используется метод `startEditing()`, за завершение редактирования отвечают `commitChanges()` и `rollback()` — однако в общем случае эти методы вам не нужны, т.к. вызываться они должны конечным пользователем.

5.6 Использование пространственного индекса

Spatial indexes can dramatically improve the performance of your code if you need to do frequent queries to a vector layer. Imagine, for instance, that you are writing an interpolation algorithm, and that for a given location you need to know the 10 closest points from a points layer, in order to use those point for calculating the interpolated value. Without a spatial index, the only way for QGIS to find those 10 points is to compute the distance from each and every point to the specified location and then compare those distances. This can be a very time consuming task, especially if it needs to be repeated for several locations. If a spatial index exists for the layer, the operation is much more effective.

Think of a layer without a spatial index as a telephone book in which telephone numbers are not ordered or indexed. The only way to find the telephone number of a given person is to read from the beginning until you find it.

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

1. create spatial index — the following code creates an empty index

   ```
   index = QgsSpatialIndex()
   ```

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

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

3. once spatial index is filled with some values, you can do some queries

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

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

5.7 Запись векторных слоёв

Для записи векторных данных на диск служит класс `QgsVectorFileWriter`. Он позволяет создавать векторные файлы в любом, поддерживаемом OGR, формате (shape-файлы, GeoJSON, KML и другие).

Существует два способа записать векторные данные в файл:

- from an instance of `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!"
  ```

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

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

- directly from features

```
# 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
```

5.8 Memory провайдер

Memory провайдер в основном предназначен для использования разработчиками расширений или сторонних приложений. Этот провайдер не хранит данные на диске, что позволят разработчикам использовать его в качестве быстрого хранилища для временных слоёв.

Провайдер поддерживает строковые и целочисленные поля, а также поля с плавающей запятой.

Memory провайдер помимо всего прочего поддерживает и пространственное индексирование, пространственный индекс можно создать вызовав функцию `createSpatialIndex()` провайдера. После создания пространственного индекса обход объектов в пределах небольшой области станет более быстрым (поскольку обращение будет идти только к объектам, попадающим в заданный прямоугольник).

Memory провайдер будет использоваться если в качестве идентификатора провайдера при вызове конструктора `QgsVectorLayer` указана строка `"memory"`.

В конструктор также передается URI, описывающий геометрию слоя, это может быть: `"Point"`, `"LineString"`, `"Polygon"`, `"MultiPoint"`, `"MultiLineString"` или `"MultiPolygon"`.

The URI can also specify the coordinate reference system, fields, and indexing of the memory provider in the URI. The syntax is:

crs=definition Задает используемую систему координат, definition может принимать любой вид, совместимый с `QgsCoordinateReferenceSystem.createFromString()`

index=yes Определяет будет ли провайдер использовать пространственный индекс

field=name:type(length,precision) Задает атрибуты слоя. Каждый атрибут имеет имя и, опционально, тип (целое число, вещественное число или строка), длину и точность. Таких описаний может быть несколько.

The following example of a URI incorporates all these options

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

The following example code illustrates creating and populating a memory provider

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

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

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

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

Finally, let's check whether everything went well

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

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

5.9 Внешний вид (символика) векторных слоёв

При отрисовке векторного слоя, внешний вид данных определяется **рендером** и **символами**, ассоциированными со слоем. Символы это классы, занимающиеся отрисовкой визуального представления объектов, а рендер опеределяет какой символ будет использован для отдельного объекта.

The renderer for a given layer can obtained as shown below:

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

And with that reference, let us explore it a bit

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

В библиотеке ядра QGIS реализовано несколько рендеров:

Тип	Класс	Описание
singleSymbol	QgsSingleSymbolRendererV2	Отрисовывает все объекты одним и тем же символом
categorizedSymbol	QgsCategorizedSymbolRendererV2	Отрисовывает объекты, используя разные символы для каждой категории
graduatedSymbol	QgsGraduatedSymbolRendererV2	Отрисовывает объекты, используя разные символы для каждого диапазона значений

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:

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

It is possible to obtain a dump of a renderer contents in text form — can be useful for debugging

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

5.9.1 Single Symbol Renderer

Получить символ, используемый для отрисовки, можно вызвав метод `symbol()`, а для его изменения служит метод `setSymbol()` (примечание для пишущих на C++: рендер становится владельцем символа).

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

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

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

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

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

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

5.9.2 Categorized Symbol Renderer

Узнать и задать поле атрибутивной таблицы, используемое для классификации можно при помощи методов `classAttribute()` и `setClassAttribute()` соответственно.

To get a list of categories

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

Здесь `value()` — величина, используемая для разделения категорий, `label()` — описание категории, а метод `symbol()` возвращает назначеный символ.

Также рендер обычно сохраняет оригинальный символ и цветовую шкалу, которые использовались для классификации, получить их можно вызвав методы `sourceColorRamp()` и `sourceSymbol()` соответственно.

5.9.3 Graduated Symbol Renderer

Этот рендер очень похож на рендер уникальными значениями, описанный выше, но вместо одного значения атрибута для класса он оперирует диапазоном значений и следовательно, может использоваться только с числовыми атрибутами.

To find out more about ranges used in the renderer

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

Как и в предыдущем случае, доступны методы `classAttribute()` для получения имени атрибута классификации, `sourceSymbol()` и `sourceColorRamp()` чтобы узнать оригинальный символ и цветовую шкалу. Кроме того, дополнительный метод `mode()` позволяет узнать какой алгоритм использовался для создания диапазонов: равные интервалы, квантили или что-то другое.

If you wish to create your own graduated symbol renderer you can do so as illustrated in the example snippet below (which creates a simple two class arrangement)

```
from qgis.core import *

myVectorLayer = QgsVectorLayer(myVectorPath, myName, 'ogr')
myTargetField = 'target_field'
myRangeList = []
myOpacity = 1
# Make our first symbol and range...
myMin = 0.0
myMax = 50.0
myLabel = 'Group 1'
myColour = QtGui.QColor('#ffee00')
mySymbol1 = QgsSymbolV2.defaultSymbol(myVectorLayer.geometryType())
mySymbol1.setColor(myColour)
mySymbol1.setAlpha(myOpacity)
myRange1 = QgsRendererRangeV2(myMin, myMax, mySymbol1, myLabel)
myRangeList.append(myRange1)
#now make another symbol and range...
myMin = 50.1
myMax = 100
myLabel = 'Group 2'
myColour = QtGui.QColor('#00eeff')
mySymbol2 = QgsSymbolV2.defaultSymbol(
     myVectorLayer.geometryType())
mySymbol2.setColor(myColour)
mySymbol2.setAlpha(myOpacity)
myRange2 = QgsRendererRangeV2(myMin, myMax, mySymbol2 myLabel)
myRangeList.append(myRange2)
myRenderer = QgsGraduatedSymbolRendererV2('', myRangeList)
```

```
myRenderer.setMode(QgsGraduatedSymbolRendererV2.EqualInterval)
myRenderer.setClassAttribute(myTargetField)

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

5.9.4 Working with Symbols

Символы представлены базовым классом `QgsSymbolV2` и тремя классами наследниками:

- `QgsMarkerSymbolV2` — for point features
- `QgsLineSymbolV2` — for line features
- `QgsFillSymbolV2` — for polygon features

Каждый символ состоит из одного и более символьных слоёв (классы, унаследованные от `QgsSymbolLayerV2`). Всю работу по отрисовке выполняют слои символа, а символ служит только контейнером для них.

Having an instance of a symbol (e.g. from a renderer), it is possible to explore it: `type()` method says whether it is a marker, line or fill symbol. There is a `dump()` method which returns a brief description of the symbol. To get a list of symbol layers

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

Узнать цвет символа можно вызвав метод `color()`, а чтобы изменить его — `setColor()`. У символов типа маркер присутствуют дополнительные методы `size()` и `angle()`, позволяющие узнать размер символа и угол поворота, а у линейных символов есть метод `width()`, возвращающий толщину линии.

Размер и толщина по умолчанию задаются в миллиметрах, а углы — в градусах.

Working with Symbol Layers

Как уже было сказано, слои символа (наследники `QgsSymbolLayerV2`) определяют внешний вид объектов. Существует несколько базовых классов символьных слоёв. Кроме того, можно создавать новые символьные слои и таким образом влиять на отрисовку объектов в достаточно широких пределах. Метод `layerType()` однозначно идентифицирует класс символьного слоя — основными и доступными по умолчанию являются символьные слои SimpleMarker, SimpleLine и SimpleFill.

You can get a complete list of the types of symbol layers you can create for a given symbol layer class like this

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

Output

```
EllipseMarker
FontMarker
SimpleMarker
SvgMarker
VectorField
```

Класс `QgsSymbolLayerV2Registry` управляет базой всех доступных символьных слоёв.

Получить доступ к данным символьного слоя можно при помощи метода `properties()`, который возвращает словарь (пары ключ-значение) свойств, влияющих на внешний вид. Символьные слои каждого типа имеют свой набор свойств. Кроме того, существуют общие для всех типов методы `color()`, `size()`, `angle()`, `width()` и соответсвующие им сеттеры. Следует помнить, что размер и угол поворота доступны только для символьных слоёв типа маркер, а толщина — только для слоёв типа линия.

Creating Custom Symbol Layer Types

Imagine you would like to customize the way how the data gets rendered. You can create your own symbol layer class that will draw the features exactly as you wish. Here is an example of a marker that draws red circles with specified radius

```
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()` определяет имя символьного слоя, которое должно быть уникальным. Чтобы все атрибуты были неизменными, используются свойства. Метод `clone()` должен возвращать копию символьного слоя с точно такими же атрибутами. И наконец, методы отрисовки: `startRender()` вызывается перед отрисовкой первого объекта, а `stopRender()` — после окончания отрисовки. За собственно отрисовку отвечает метод `renderPoint()`. Координаты точки (точек) должны быть трансформирваны в выходные координаты.

Для полининий и полигонов единственное отличие будет в методе отрисовки: необходимо использовать `renderPolyline()`, принимающий список линий, или `renderPolygon()` в качестве первого аргумента принимающий список точек, образующих внешнее кольцо, и список внутренних колец (или None) вторым аргументом.

Usually it is convenient to add a GUI for setting attributes of the symbol layer type to allow users to customize the appearance: in case of our example above we can let user set circle radius. The following code implements such widget

```
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()"))
```

Этот виджет можно встроить в диалог свойств символа. Когда символьный слой выделяется в диалоге свойств символа, создается экземпляр символьного слоя и экземпляр виджета символьного слоя. Затем вызывается метод `setSymbolLayer()` чтобы привязать символьный слой к виджету. В этом методе виджет должен обновить интерфейс, чтобы отразить атрибуты символьного слоя. Функция `symbolLayer()` используется диалогом свойств для получения измененного символьного слоя для дальнейшего использования.

При каждом изменении атрибутов виджет должен посылать сигнал `changed()`, чтобы диалог свойств мог обновить предпросмотр символа.

Остался последний штрих: рассказать QGIS о существовании этих новых классов. Для этого достаточно добавить символьный слой в реестр. Конечно, можно использовать символьный слой и не добавляя его в реестр, но тогда некоторые возможности будут недоступны: например, загрузка проекта с пользовательскими символьными слоями или невозможность редактировать свойства слоя.

We will have to create metadata for the symbol layer

```
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())
```

В конструктор родительского класса необходимо передать тип слоя (тот же, что сообщает слой) и тип символа (маркер/линия/заливка). `createSymbolLayer()` создаёт экземпляр символьного слоя с атрибутами, указаными в словаре *props*. (Будьте внимательны, ключи являются экземплярами QString, а не объектами "str"). Метод `createSymbolLayerWidget()` должен возвращать виджет настроек этого символьного слоя.

Последней конструкцией мы добавляем символьный слой в реестр — на этом все.

5.9.5 Creating Custom Renderers

Возможность создать свой рендер может быть полезной, если требуется изменить правила выбора символов для отрисовки объектов. Примерами таких ситуаций могут быть: символ должен определяться на основании значений нескольких полей, размер символа должен зависеть от текущего масштаба и т.д.

The following code shows a simple custom renderer that creates two marker symbols and chooses randomly one of them for every feature

```
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)
```

В конструктор родительского класса `QgsFeatureRendererV2` необходимо передать имя ренедера (должно быть уникальным). Метод `symbolForFeature()` определяет какой символ будет использоваться для конкретного объекта. `startRender()` и `stopRender()` выполняют инициализацию/финализацию отрисовки символа. Метод `usedAttributes()` может возвращать список имен полей, которые необходимы рендеру. И, наконец, функция `clone()` должна возвращать копию рендера.

Like with symbol layers, it is possible to attach a GUI for configuration of the renderer. It has to be derived from `QgsRendererV2Widget`. The following sample code creates a button that allows user to set symbol of the first symbol

```
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
```

В конструктор передается экземпляры активного слоя (`QgsVectorLayer`), глобальный стиль (`QgsStyleV2`) и текущий рендер. Если рендер не задан или имеет другой тип, он будет заменен нашим рендером, в противном случае мы будем использовать текущий рендер (который нам и нужен). Необходимо обновить содержимое виджета, чтобы отразить текущее состояние рендера. При закрытии диалога ренедера, вызывается метод `renderer()` виджета чтобы получить текущий рендер — он будет назначен слою.

The last missing bit is the renderer metadata and registration in registry, otherwise loading of layers with the renderer will not work and user will not be able to select it from the list of renderers. Let us finish our RandomRenderer example

```
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())
```

Так же, как и в случае символьных слоёв, абстрактный конструктор метаданных должен получить имя рендера, отображаемое имя и, по желанию, название иконки рендера. Метод `createRenderer()` получает экземпляр `QDomElement`, который может использоваться для восстановления состояния рендера из дерева DOM. Метод `createRendererWidget()` отвечает за создание виджета настройки. Он может отсутствовать или возвращать *None*, если рендер не имеет интрерфейса.

To associate an icon with the renderer you can assign it in `QgsRendererV2AbstractMetadata` constructor as a third (optional) argument — the base class constructor in the RandomRendererMetadata `__init__()` function becomes

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

Иконку можно назначить и позже, воспользовавшись методом `setIcon()` класса метаданных. Иконка может загружаться из файла (как показано выше) или из ресурсов Qt (в составе PyQt4 присутствует компилятор .qrc для Python).

5.10 Further Topics

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

Обработка геометрии

Points, linestrings and polygons that represent a spatial feature are commonly referred to as geometries. In QGIS they are represented with the `QgsGeometry` class. All possible geometry types are nicely shown in JTS discussion page.

Sometimes one geometry is actually a collection of simple (single-part) geometries. Such a geometry is called a multi-part geometry. If it contains just one type of simple geometry, we call it multi-point, multi-linestring or multi-polygon. For example, a country consisting of multiple islands can be represented as a multi-polygon.

Координаты, описывающие геометрию, могут быть в любой системе координат (CRS). Когда выполняется доступ к объектам слоя, ассоциированые геометрии будут выданы с координатами в СК слоя.

6.1 Создание геометрий

There are several options for creating a geometry:

- 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)]])
  ```

  Координаты задаются при помощи класса `QgsPoint`.

  Полилиния описывается массивом точек. Полигон представляется как список линейных колец (например, замкнутых линий). Первое кольцо — внешнее (граница), последующие не обязательные кольца описывают дырки в полигоне.

  Составные геометрии имеют дополнительный уровень вложености, так: мульти-точка это список точек, мульти-линия — список линий и мульти-полигон является списком полигонов.

- from well-known text (WKT)

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

- from well-known binary (WKB)

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

6.2 Доступ к геометрии

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.

To extract information from geometry there are accessor functions for every vector type. How to use accessors

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

Примечание: очередь (x,y) не является настоящей очередью, это объект `QgsPoint`, а к его значениям можно обратиться при помощи методов `x()` и `y()`.

Для составных геометрий существуют аналогичныей фукнции доступа: `asMultiPoint()`, `asMultiPolyline()`, `asMultiPolygon()`.

6.3 Геометрические предикаты и операции

QGIS uses GEOS library for advanced geometry operations such as geometry predicates (`contains()`, `intersects()`, ...) and set operations (`union()`, `difference()`, ...). It can also compute geometric properties of geometries, such as area (in the case of polygons) or lengths (for polygons and lines)

Here you have a small example that combines iterating over the features in a given layer and performing some geometric computations based on their geometries.

```
# 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()
```

Areas and perimeters don't take CRS into account when computed using these methods from the `QgsGeometry` class. For a more powerful area and distance calculation, the `QgsDistanceArea` class can be used. If projections are turned off, calculations will be planar, otherwise they'll be done on the ellipsoid. When an ellipsoid is not set explicitly, WGS84 parameters are used for calculations.

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

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

You can find many example of algorithms that are included in QGIS and use these methods to analyze and transform vector data. Here are some links to the code of a few of them.

Additional information can be found in following sources:

- Geometry transformation: Reproject algorithm
- Distance and area using the `QgsDistanceArea` class: Distance matrix algorithm

- Multi-part to single-part algorithm

Поддержка проекций

7.1 Coordinate reference systems

Coordinate reference systems (CRS) are encapsulated by `QgsCoordinateReferenceSystem` class. Instances of this class can be created by several different ways:

- specify CRS by its ID

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

  QGIS uses three different IDs for every reference system:

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

  If not specified otherwise in second parameter, PostGIS SRID is used by default.

- 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")
  ```

It's wise to check whether creation (i.e. lookup in the database) of the CRS has been successful: `isValid()` must return `True`.

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

Accessing spatial reference system information

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

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

7.2 Projections

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

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

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

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

Глава 8

Работа с картой

Виджет “карта” (Map Canvas) является одним из наиболее важных, так как именно он отвечает за отображение карты, состоящей из наложенных друг на друга слоёв, и позволяет взаимодействовать как со всей картой, так и с отдельными слоями. Виджет отображает только часть карты, заданную текущим охватом. Взаимодействие выполняется при помощи **инструментов карты** (map tools): среди которых присутствуют инструменты панорамирования, масштабирования, определения слоёв, измерения, редактирования и другие. Как и в других программах, активным в каждый момент времени может быть только один инструмент, при необходимости выполняется переключение между 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.

Всякий раз, когда пользователь выполняет панорамирование, масштабирование (или любое другое действие, вызывающее обновление карты), происходит перерисовка карты в пределах текущего охвата. Отрисовка слоёв выполняется в изображение (за это отвечает класс `QgsMapRenderer`), которое затем отображается на карте. Графическим объектом (в терминах фреймвока Qt — graphics view), отвечающим за отображение карты, является класс `QgsMapCanvasMap`. Этот же класс следит за обновлением карты. Помимо этого объекта, который служит фоном, может существовать множество **элементов карты**. Обычно, в роли элементов карты выступают “резиновые” линии (используемые при измерении и редактировании слоёв) или маркеры вершин. Чаще всего элементы карты используются для визуализации работы инструментов карты. Например, при создании нового полигона, инструмент карты создает “резиновый” элемент карты, показывающий текущую форму полигона. Все элементы карты являются наследниками `QgsMapCanvasItem` и добавляют свой функционал к базовому объекту `QGraphicsItem`.

Таким образом, архитектурно карта состоит из трёх элементов:

- карта — для отображения данных
- элементы карты — дополнительные объекты, которые можно отобразить на карте
- инструменты карты — обеспечивают взаимодействие с картой

8.1 Встраивание карты

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

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

Этот код создаст новое окно с картой. Точно так же можно встраивать карту в существующий виджет или окно. При использовании Qt Designer и файлов .ui удобно делать так: на форму положить `QWidget` и объявить его новым классом, установив в качестве имени класса `QgsMapCanvas` и `qgis.gui` в качестве заголовочного файла. Всё остальное сделает программа `pyuic4`. Как видите, это очень простой и удобный способ встраивания карты в приложение. Ещё один способ — создать виджет карты и другие элементы интерфейса динамически (в качестве дочерних объектов основного или диалогового окна) и разместить их на компоновке.

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

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

(если вас интересует, то приставка `Qt` используется модулем `PyQt4.QtCore` а `Qt.white` это один из предварительно заданных экземпляров `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)])
```

После выполнения этих команд на карте должен отобразиться загруженный слой.

8.2 Использование инструментов карты

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

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

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

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

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

    self.setCentralWidget(self.canvas)

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

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

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

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

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

    self.pan()

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

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

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

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

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

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

8.3 Резиновые полосы и маркеры 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.

To show a polyline

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

To show a polygon

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

Обратите внимание, что узлы полигона представлены не плоским списком: на самом деле это список границ полигона. Первое кольцо описывает внешний контур, все остальные (не обязательные) — соответствуют дыркам в полигоне.

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

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

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

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

(при использовании C++ можно просто удалить элемент, однако в Python `del r` удалит только ссылку, а сам объект останется на месте, т.к. его владельцем является карта)

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

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

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

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

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

8.4 Создание собственных инструментов карты

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

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

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

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

  def reset(self):
```

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

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

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

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

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

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

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

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

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

        return QgsRectangle(self.startPoint, self.endPoint)

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

8.5 Создание собственных элементов карты

TODO: how to create a map canvas item

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

def init():
  a = QgsApplication(sys.argv, True)
```

```
  QgsApplication.setPrefixPath('/home/martin/qgis/inst', True)
  QgsApplication.initQgis()
  return a

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

Глава 9

Отрисовка карты и печать

Существует два способа получить печатную карту из исходных данных: простой и быстрый используя `QgsMapRenderer` или создание и тщательная настройка компоновки используя `QgsComposition` и сопутствующие классы.

9.1 Простая отрисовка

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

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

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

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

render = QgsMapRenderer()

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

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

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

# do the rendering
render.render(p)

p.end()

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

9.2 Rendering layers with different CRS

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

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

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

9.3 Вывод с использованием компоновщика карт

Компоновщик карт это удобный инструмент для создания более сложных печатных карт, по сравнению с простой отрисовкой описанной выше. Используя компоновщик можно создавать составные компоновки, содержащие карты, подписи, легенду, таблицы и другие элементы, которые обычно присутсвуют на печатных картах. Готовую компоновку можно экспортировать в PDF, растровое изображение или сразу же распечатать на принтере.

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

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

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

Обратите вниманием, что компоновка принимает в качестве параметра экземпляр `QgsMapRenderer`. Предполагается, что код выполняется в среде QGIS и поэтому используется рендер активной карты. Компоновка использует различные параметры рендера, и самое главное — набор слоёв карты и текущий охват. При использовании компоновщика в самостоятельном приложении необходимо создать свой собственный экземпляр рендера, как это показано в разделе выше, и передать его в компоновку.

К компоновке можно добавлять разные элементы (карту, подписи, ...) — все они являются потомками класса `QgsComposerItem`. В настоящее время доступны следующие элементы:

- 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)
```

- стрелка севера
- изображение
- фигура
- таблица

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

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

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

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

Помимо создания элементов компоновки вручную QGIS поддерживает шаблоны компоновок, которые являются компоновками со всеми элементами, сохраненными в файл .qpt (формат XML). К сожалению, этот функционал пока ещё не доступен в API.

После того как компоновка готова (все элементы созданы и добавлены к компоновке), можно приступать к представлению результатов в растровой или векторной форме.

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

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

9.3.1 Вывод в растровое изображение

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

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

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

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

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

9.3.2 Вывод в формате PDF

The following code fragment renders a composition to a PDF file

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

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

Глава 10

Выражения, фильтрация и вычисление значений

QGIS может выполнять разбор и анализ SQL-подобных выражений. Поддерживается ограниченное подмножество языка SQL. Выражения могут рассматриваться как логические предикаты (возвращающие True или False) или как функции (возвращающие скалярное значение).

Поддерживается три основных типа данных:

- число — как целые, так и десятичные, например, `123`, `3.14`
- строка — должна заключаться в одинарные кавычки: `'hello world'`
- ссылка на столбец — при вычислении ссылка заменяется на значение поля. Имена полей не экранируются.

Доступны следующие операции:

- арифметические операторы: `+`, `-`, `*`, `/`, `^`
- скобки: для изменения приоритета операций: `(1 + 1) * 3`
- унарный плюс и минус: `-12`, `+5`
- математические функции: `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
- геометрические функции: `$area`, `$length`
- функции преобразования типа: `to int`, `to real`, `to string`

And the following predicates are supported:

- сравнение: `=`, `!=`, `>`, `>=`, `<`, `<=`
- соответствие образцу: `LIKE` (using % and _), `~` (регулярные выражения)
- огические операторы: `AND`, `OR`, `NOT`
- проверка на NULL: `IS NULL`, `IS NOT NULL`

Примеры предикатов:

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

Примеры скалярных выражений:

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

10.1 Разбор выражений

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

10.2 Вычисление выражений

10.2.1 Простые выражения

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

10.2.2 Выражения с объектами

The following example will evaluate the given expression against a feature. "Column" is the name of the field in the layer.

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

You can also use `QgsExpression.prepare()` if you need check more than one feature. Using `QgsExpression.prepare()` will increase the speed that evaluate takes to run.

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

10.2.3 Handling errors

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

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

print value
```

10.3 Примеры

Следующие примеры могут использоваться для фильтрации слоя и возвращения объектов, удовлетворяющих условию.

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

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

Глава 11

Чтение и сохранение настроек

Часто бывает полезным сохранить некоторые параметры расширения, чтобы пользователю не приходилось заново вводить или выбирать их при каждом запуске расширения.

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

Необходимо различать следующие типы настроек:

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

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

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

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

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

```
proj = QgsProject.instance()

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

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

As you can see, the `writeEntry()` method is used for all data types, but several methods exist for reading the setting value back, and the corresponding one has to be selected for each data type.

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

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

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

Глава 12

Communicating with the user

This section shows some methods and elements that should be used to communicate with the user, in order to keep consistency in the User Interface.

12.1 Showing messages. The `QgsMessageBar` class

Using message boxes can be a bad idea from a user experience point of view. For showing a small info line or a warning/error messages, the QGIS message bar is usually a better option.

Using the reference to the QGIS interface object, you can show a message in the message bar with the following code

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

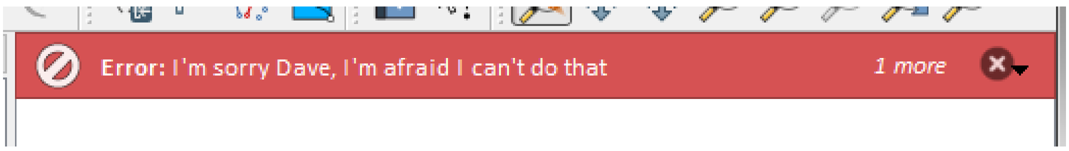

Рис. 12.1: QGIS Message bar

You can set a duration to show it for a limited time

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

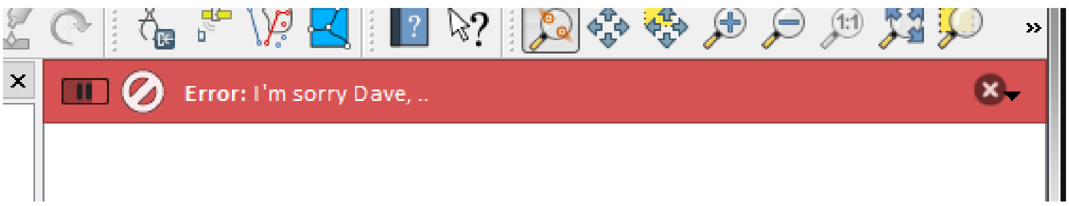

Рис. 12.2: QGIS Message bar with timer

The examples above show an error bar, but the `level` parameter can be used to creating warning messages or info messages, using the `QgsMessageBar.WARNING` and `QgsMessageBar.INFO` constants respectively.

Widgets can be added to the message bar, like for instance a button to show more info

```
def showError():
    pass

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

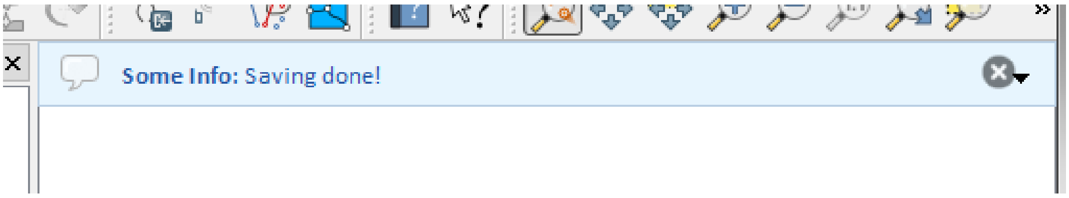

Рис. 12.3: QGIS Message bar (info)

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

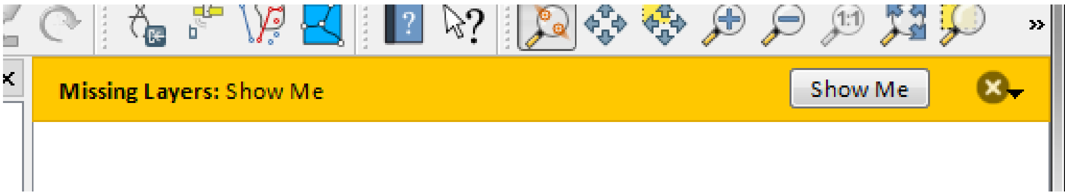

Рис. 12.4: QGIS Message bar with a button

You can even use a message bar in your own dialog so you don't have to show a message box, or if it doesn't make sense to show it in the main QGIS window

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

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

12.2 Showing progress

Progress bars can also be put in the QGIS message bar, since, as we have seen, it accepts widgets. Here is an example that you can try in the console.

```
import time
from PyQt4.QtGui import QProgressBar
from PyQt4.QtCore import *
progressMessageBar = iface.messageBar().createMessage("Doing something boring...")
progress = QProgressBar()
progress.setMaximum(10)
progress.setAlignment(Qt.AlignLeft|Qt.AlignVCenter)
progressMessageBar.layout().addWidget(progress)
iface.messageBar().pushWidget(progressMessageBar, iface.messageBar().INFO)
for i in range(10):
    time.sleep(1)
```

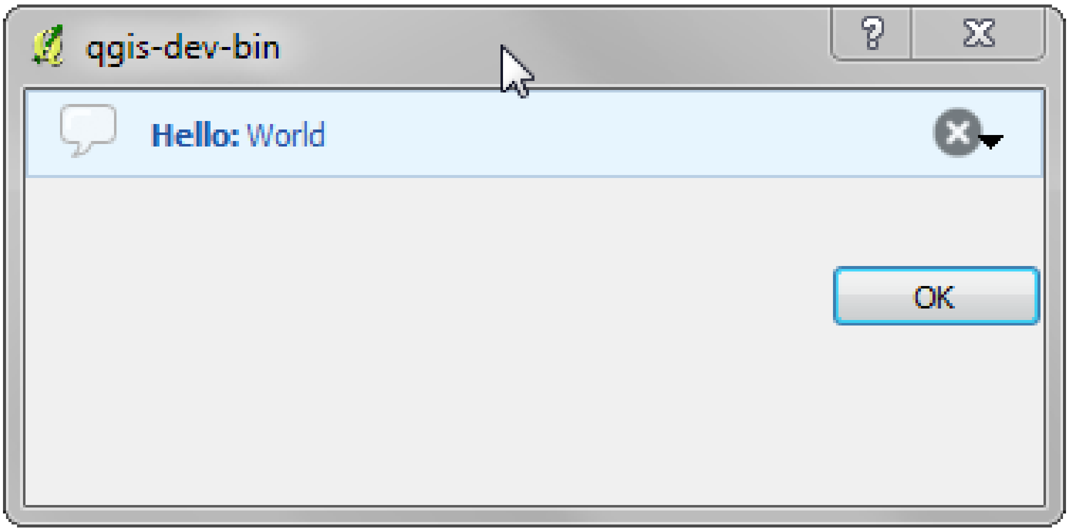

Рис. 12.5: QGIS Message bar in custom dialog

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

Also, you can use the built-in status bar to report progress, as in the next example

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

12.3 Logging

You can use the QGIS logging system to log all the information that you want to save about the execution of your code.

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

Глава 13

Разработка расширений на Python

Для разработки расширений можно использовать язык программирования Python. По сравнению с классическими расширениями, написанными на C++, их легче разрабатывать, понимать, поддерживать и распространять в силу динамической природы самого Python.

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

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

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

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

Шаги:

1. *Идея*: Прежде всего нужна идея для нового расширения QGIS. Зачем это нужно? Какую задачу необходимо решить? Может, есть готовое расширения для решения этой задачи?
2. *Create files*: Create the files described next. A starting point (`__init__.py`). Fill in the *Plugin metadata* (`metadata.txt`) A main python plugin body (`mainplugin.py`). A form in QT-Designer (`form.ui`), with its `resources.qrc`.
3. *Write code*: Write the code inside the `mainplugin.py`
4. *Тестирование*: Закройте и снова откройте QGIS, загрузите своё расширение. Проверьте, что всё работает как надо.
5. *Publish*: Publish your plugin in QGIS repository or make your own repository as an "arsenal" of personal "GIS weapons".

13.1 Разработка расширения

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

13.1.1 Plugin files

Here's the directory structure of our example plugin

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

Для чего используются файлы:

- `__init__.py` = The starting point of the plugin. It has to have the `classFactory()` method and may have any other initialisation code.
- `mainPlugin.py` = The main working code of the plugin. Contains all the information about the actions of the plugin and the main code.
- `resources.qrc` = The .xml document created by Qt Designer. Contains relative paths to resources of the forms.
- `resources.py` = The translation of the .qrc file described above to Python.
- `form.ui` = The GUI created by Qt Designer.
- `form.py` = The translation of the form.ui described above to Python.
- `metadata.txt` = Required for QGIS >= 1.8.0. Containts general info, version, name and some other metadata used by plugins website and plugin infrastructure. Since QGIS 2.0 the metadata from `__init__.py` are not accepted anymore and the `metadata.txt` is required.

Here is an online automated way of creating the basic files (skeleton) of a typical QGIS Python plugin.

Also there is a QGIS plugin called Plugin Builder that creates plugin template from QGIS and doesn't require internet connection. This is the recommended option, as it produces 2.0 compatible sources.

Предупреждение: If you plan to upload the plugin to the *Official python plugin repository* you must check that your plugin follows some additional rules, required for plugin *Validation*

13.2 Plugin content

Here you can find information and examples about what to add in each of the files in the file structure described above.

13.2.1 Plugin metadata

First, plugin manager needs to retrieve some basic information about the plugin such as its name, description etc. File `metadata.txt` is the right place to put this information.

Важно: All metadata must be in UTF-8 encoding.

Metadata name	Required	Notes
имя	True	a short string containing the name of the plugin
qgisMinimumVersion	True	dotted notation of minimum QGIS version
qgisMaximumVersion	False	dotted notation of maximum QGIS version
description	True	short text which describes the plugin, no HTML allowed
about	False	longer text which describes the plugin in details, no HTML allowed
version	True	short string with the version dotted notation
author	True	author name
email	True	email of the author, will *not* be shown on the web site
changelog	False	string, can be multiline, no HTML allowed
experimental	False	boolean flag, *True* or *False*
deprecated	False	boolean flag, *True* or *False*, applies to the whole plugin and not just to the uploaded version
tags	False	comma separated list, spaces are allowed inside individual tags
homepage	False	a valid URL pointing to the homepage of your plugin
repository	False	a valid URL for the source code repository
tracker	False	a valid URL for tickets and bug reports
icon	False	a file name or a relative path (relative to the base folder of the plugin's compressed package)
category	False	one of *Raster*, *Vector*, *Database* and *Web*

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

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

```
category=Raster
```

Примечание: If *qgisMaximumVersion* is empty, it will be automatically set to the major version plus *.99* when uploaded to the *Official python plugin repository*.

An example for this metadata.txt

```
; the next section is mandatory

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

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

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

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

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

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

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

13.2.2 __init__.py

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

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

## any other initialisation needed
```

13.2.3 mainPlugin.py

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

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

# initialize Qt resources from file resources.py
import resources

class TestPlugin:

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

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

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

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

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

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

All of them have the same syntax as the `addPluginToMenu()` method.

Adding your plugin menu to one of those predefined method is recommended to keep consistency in how plugin entries are organized. However, you can add your custom menu group directly to the menu bar, as the next example demonstrates:

```
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.

13.2.4 Файл ресурсов

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

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

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

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

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

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

При работе над реальным расширением удобно вести разработку в другом (рабочем) каталоге и создать makefile, который будет генерировать файлы интерфейса и ресурсов, а также выполнять копирование расширения в каталог QGIS.

13.3 Документация

The documentation for the plugin can be written as HTML help files. The qgis.utils module provides a function, showPluginHelp() which will open the help file browser, in the same way as other QGIS help.

Функция showPluginHelp‘() ищет файлы справки в том же каталоге, где находится вызвавший её модуль. Она по очереди будет искать файлы index-ll_cc.html, index-ll.html, index-en.html, index-en_us.html и index.html, и отобразит первый найденный. Здесь ll_cc — язык интерфейса QGIS. Это позволяет включать в состав расширения документацию на разных языках.

Кроме того, функция showPluginHelp() может принимать параметр packageName, идентифицирующий расширение, справку которого нужно отобразить; filename, который используется для переопределения имени файла с документацией; и section, для передачи имени якоря (закладки) в документе, на который браузер должен перейти.

Глава 14

IDE settings for writing and debugging plugins

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

14.1 A note on configuring your IDE on Windows

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

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

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

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

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

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

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

To create that batch file, follow these steps.

- Locate the folder where file:*qgis_core.dll* resides in. Normally this is `C:OSGeo4Wappsqgisbin`, but if you compiled your own QGIS application this is in your build folder in `output/bin/RelWithDebInfo`
- Locate your `eclipse.exe` executable.
- Create the following script and use this to start eclipse when developing QGIS plugins.

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

14.2 Debugging using Eclipse and PyDev

14.2.1 Installation

To use Eclipse, make sure you have installed the following

- Eclipse
- Aptana Eclipse Plugin or PyDev
- QGIS 2.0

14.2.2 Preparing QGIS

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

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

14.2.3 Setting up Eclipse

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

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

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

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

14.2.4 Configuring the debugger

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

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

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

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

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

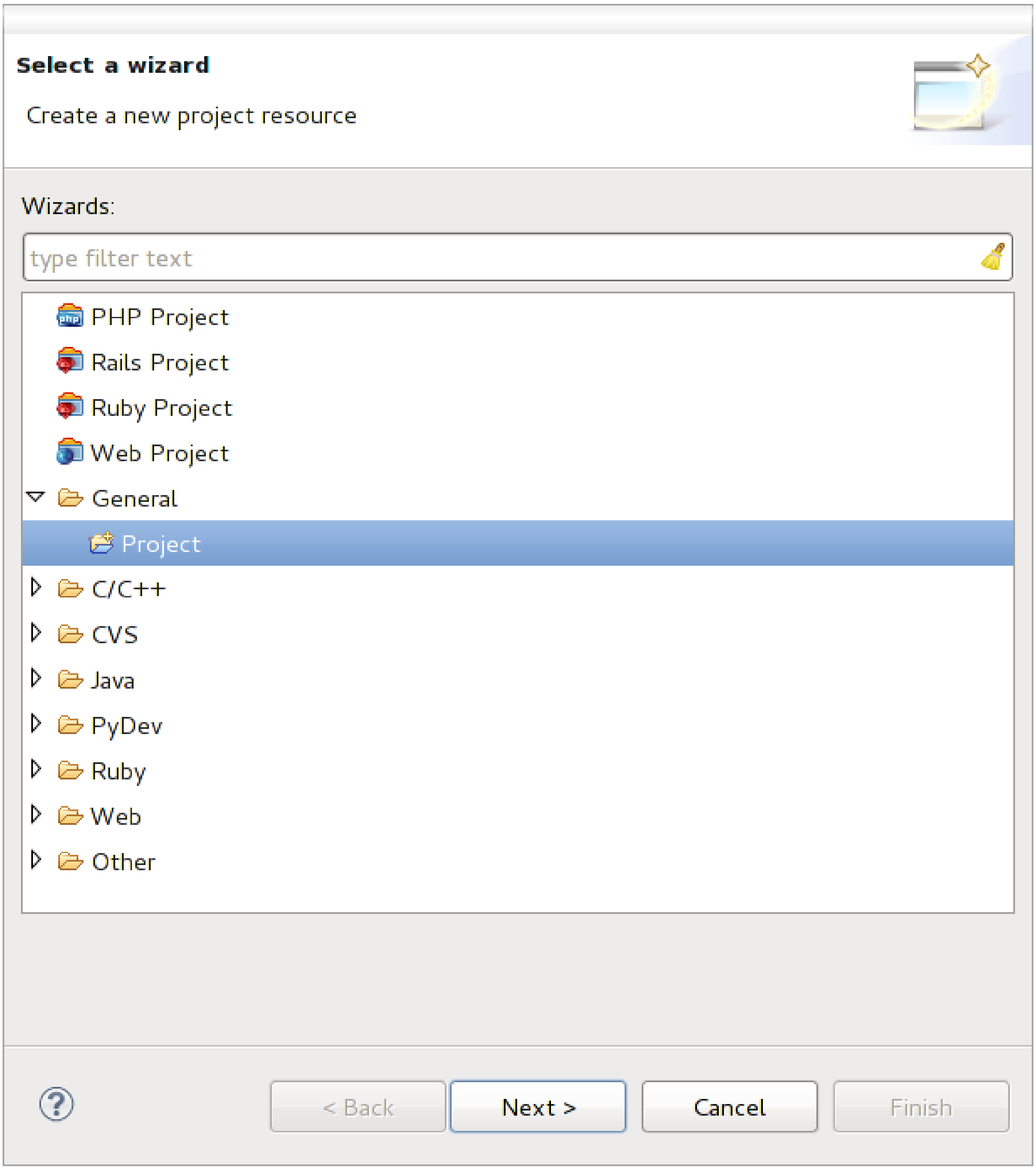

Рис. 14.1: Eclipse project

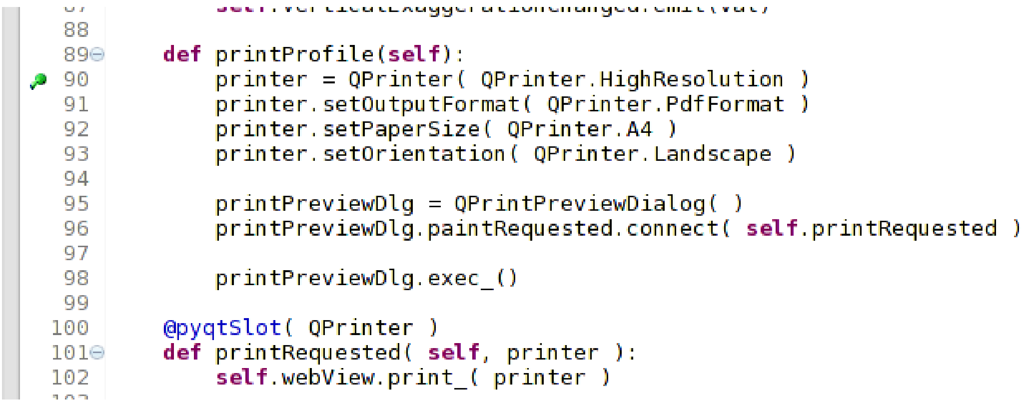

Рис. 14.2: Breakpoint

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

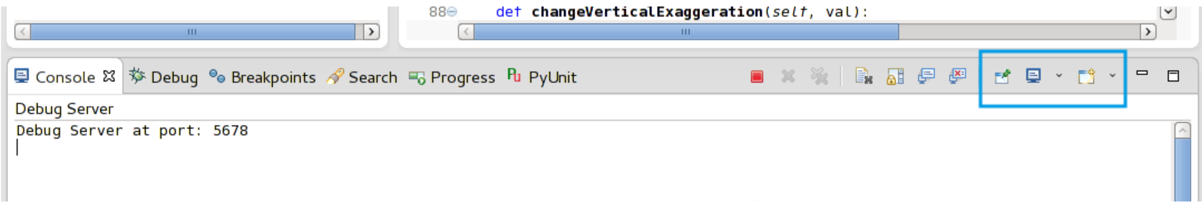

Рис. 14.3: PyDev Debug Console

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

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

14.2.5 Making eclipse understand the API

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

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

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

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

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

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

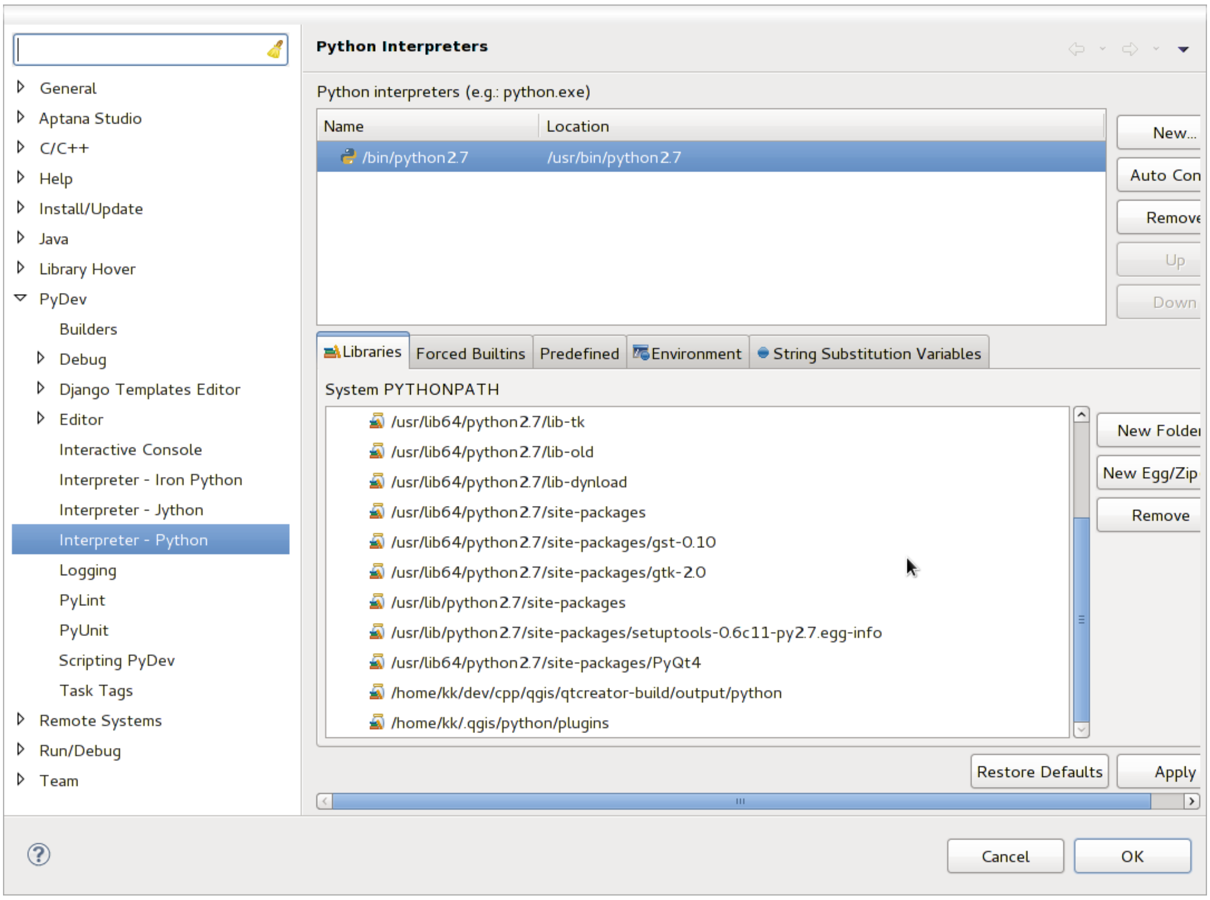

Рис. 14.4: PyDev Debug Console

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

Click *OK* and you're done.

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

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

14.3 Debugging using PDB

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

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

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

Then run QGIS from the command line.

On Linux do:

$ ./Qgis

On Mac OS X do:

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

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

TODO: Add testing information

Использование слоёв расширений

Если расширение использует собственные методы для отрисовки слоёв карты, наиболее простой способ реализации — создание нового типа слоя на основе QgsPluginLayer.

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

15.1 Наследование 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
```

Methods for reading and writing specific information to the project file can also be added

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

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

When loading a project containing such a layer, a factory class is needed

```
class WatermarkPluginLayerType(QgsPluginLayerType):

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

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

You can also add code for displaying custom information in the layer properties

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

Глава 16

Совместимость с ранними версиями QGIS

16.1 Plugin menu

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

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

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

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

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

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

Глава 17

Releasing your plugin

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

Please take special care to the following suggestions:

17.1 Metadata and names

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

17.2 Code and help

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

17.3 Official python plugin repository

You can find the *official* python plugin repository at http://plugins.qgis.org/.

In order to use the official repository you must obtain an OSGEO ID from the OSGEO web portal.

Once you have uploaded your plugin it will be approved by a staff member and you will be notified.

TODO: Insert a link to the governance document

17.3.1 Permissions

These rules have been implemented in the official plugin repository:

- every registered user can add a new plugin
- *staff* users can approve or disapprove all plugin versions
- users which have the special permission *plugins.can_approve* get the versions they upload automatically approved
- users which have the special permission *plugins.can_approve* can approve versions uploaded by others as long as they are in the list of the plugin *owners*
- a particular plugin can be deleted and edited only by *staff* users and plugin *owners*
- if a user without *plugins.can_approve* permission uploads a new version, the plugin version is automatically unapproved.

17.3.2 Trust management

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

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

17.3.3 Validation

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

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

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

17.3.4 Plugin structure

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

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

Code Snippets

This section features code snippets to facilitate plugin development.

18.1 How to call a method by a key shortcut

In the plug-in add to the `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)
```

To `unload()` add

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

The method that is called when F7 is pressed

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

18.2 How to toggle Layers

Since QGIS 2.4 there is new layer tree API that allows direct access to the layer tree in the legend. Here is an example how to toggle visibility of the active layer

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

18.3 How to access attribute table of selected features

```
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")
```

The method requires one parameter (the new value for the attribute field of the selected feature(s)) and can be called by

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

Библиотека сетевого анализа

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

- может создавать математический граф из географических данных (линейных векторных слоёв)
- implements basic methods from graph theory (currently only Dijkstra's algorithm)

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

19.1 Применение

Briefly, a typical use case can be described as:

1. получить граф из географических данных
2. выполнить анализ графа
3. использовать результаты анализа (например, визуализировать их)

19.2 Building a graph

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

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

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

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

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

To calculate edge properties the programming pattern strategy is used. For now only QgsDistanceArcProperter strategy is available, that takes into account the length of the route. You can implement your own strategy that will use all necessary parameters. For example, RoadGraph plugin uses a strategy that computes travel time using edge length and speed value from attributes.

It's time to dive into the process.

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

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

Then some examples for creating a director

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

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

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

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

And here is full list of what these parameters mean:

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

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

```
properter = QgsDistanceArcProperter()
```

And tell the director about this strategy

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

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

- `crs` — используемая система координат. Обязательный параметр.
- `otfEnabled` — указывает на использование перепроецирования «на лету». По умолчанию True

- `topologyTolerance` — топологическая толерантность. Значение по умолчанию 0
- *ellipsoidID‘* — используемый эллипсоид. По умолчанию “WGS84”

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

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

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

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

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

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

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

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

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

19.3 Анализ графа

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

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

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

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

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

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

- `source` — исходный граф
- `startVertexIdx` — индекс точки на графе (корень дерева)
- `criterionNum` — порядковый номер свойства ребра (отсчет ведется от 0)

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

Метод `dijkstra()` имеет аналогичные параметры, но возвращает не граф, а кортеж из двух массивов. В первом массиве `i`-ый элемент содержит индекс дуги, входящей в `i`-ю вершину, в противном случае — -1. Во втором массиве `i`-ый элемент содержит расстояние от корня дерева до `i`-ой вершины, если вершина достижима из корня или максимально большое число которое может хранить тип C++ double, если вершина не достижима.

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

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

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

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

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

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

graph = builder.graph()

idStart = graph.findVertex(pStart)

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

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

Same thing but using `dijkstra()` method

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

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

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

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

graph = builder.graph()

idStart = graph.findVertex(pStart)
```

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

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

19.3.1 Finding shortest paths

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

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

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

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

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

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

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

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

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

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

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

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

if idStop == -1:
```

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

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

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

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

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

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

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

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

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

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

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

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

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

  p.append(tStart)

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

  for pnt in p:
```

```
    rb.addPoint(pnt)
```

19.3.2 Areas of availability

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

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

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

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

Here is an example

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

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

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

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

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

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

idStart = graph.findVertex(tStart)

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

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

```
  i = i + 1

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