# Документация PyQGIS

Martin Dobias

Перевод: Александр Бруй (NextGIS)
Обновления: GIS-Lab.info

18 April 2012

# Оглавление

Содержание:

# Введение

Этот документ задумывался как учебник и справочное пособие. Вы не найдете 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:

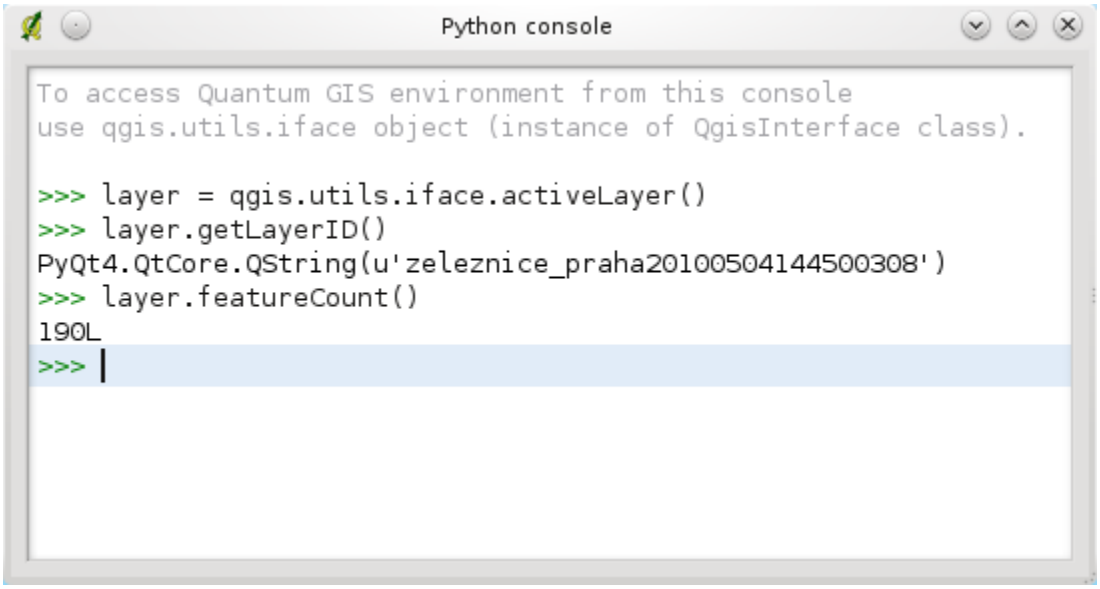
- ввод команд в консоли Python QGIS
- создание и использование расширений на Python
- создание собственного приложения на базе QGIS API

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

Кроме того, некоторая информация о разработке с использованием PyQGIS есть в блоге QGIS. Например, в записи QGIS tutorial ported to Python приведены примеры самостоятельных приложений. Хороший способ узнать о работе с расширениями — загрузить несколько готовых модулей из репозитория и изучить их код.

## 1.1 Консоль Python

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




На рисунке показано как можно получить выделенный в окне легенды слой, отобразить его ID и, например, если это векторный слой, узнать количество объектов. Для взаимодействия с QGIS предназначена переменная qgis.utils.iface, которая является экземпляром класса QgisInterface. Используя этот интерфейс можно обращаться к карте, меню, панелям инструментов и другим частям QGIS.

Для удобства пользователей при открытии консоли выполняются следующие команды (в дальнейшем можно будет расширять этот список):

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

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

## 1.2 Расширения на Python

Quantum GIS позволяет расширять свой функционал при помощи расширений. Изначально это было возможно только с использованием языка программирования C++. Позже, когда в QGIS появилась поддержка Python, стало возможным создание и использование расширений на Python. Их большим преимуществом, по сравнению с расширениями на C++, является простота распространения (не требуется компиляция под разные платформы) и легкость разработки.

С момента введения поддержки Python было разработано множество расширений, добавляющих самые разные функциии. Установщик модулей позволяет пользователям легко получать, обновлять и удалять расширения на Python. Обратитесь к странице Python Plugin Repositories для получения дополнительной информации.

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

## 1.3 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.

### 1.3.1 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 *

# подставьте путь к папке, где установлена QGIS
QgsApplication.setPrefixPath("/path/to/qgis/installation", True)

# загрузка провайдеров
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()
```

### 1.3.2 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.

# 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.

## 2.1 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.

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

- библиотека OGR (shape-файлы и другие форматы) — идентификатором источника данных является путь к файлу:

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

- база данных PostGIS — идентификатором источника данных выступает строка с информацией, необходимой для установки соединения с базой данных PostgreSQL. Используйте класс QgsDataSourceURI для формирования такой строки. Помните, что QGIS должна быть скомпилированна с поддержкой PostgreSQL, иначе этот провайдер будет недоступен.

  ```
  uri = QgsDataSourceURI()
  # задаем имя хоста, порт, название базы данных, имя пользователя и пароль
  uri.setConnection("localhost", "5432", "dbname", "johny", "xxx")
  # задается схема, название таблицы, поле геометрии и, опционально, подмножество (условие WHERE)
  uri.setDataSource("public", "roads", "the_geom", "cityid = 2643")

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

- CSV или другие текстовые файлы с разделителями — для открытия файла с полями “x” для координаты X и “y” для координаты Y и использующего в качестве разделителя запятую, можно использовать такой код:

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

  Примечание: начиная с QGIS 1.7 строка вызова провайдера формируется в виде URL, поэтому путь должен начинаться с file:///. Кроме того, допускается использование геометрии в формате WKT (well known text) вместо полей с координатами x и y, и допускатся указание желаемой системы координат. Например:

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

- GPX файлы — провайдер данных “gpx” позволяет читать треки, маршруты и путевые точки из файлов gpx. При открытии файла необходимо указать его тип (track/route/waypoint) в качестве части url:

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

- база данных SpatiaLite — поддерживается начиная с QGIS v1.1. Как и в случае с базами PostGIS, для генерирования идентификатора источника данных можно использовать QgsDataSourceURI:

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

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

- WKB-геометрия из базы MySQL, доступ выполняется при помощи OGR — в качестве идентификатора источника даных выступает строка подключения к таблице:

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

## 2.2 Растровые слои

Для работы с растровыми данными используется библиотека GDAL. Она поддерживает множество различных форматов. В случае проблем при открытии файлов проверьте поддерживает ли ваша версия GDAL этот формат (поддержка некоторых форматов по умолчанию не доступна). Для загрузки растра из файла необходимо указать его имя и имя файла:

```
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!"
```

Или же можно загрузить растровый слой с сервера WMS. Однако, в настоящее время в API не предусмотрена возможность получить доступ к ответу на запрос GetCapabilities — необходимо знать названия нужных слоёв:

```
url = 'http://wms.jpl.nasa.gov/wms.cgi'
layers = [ 'global_mosaic' ]
```

```
styles = [ 'pseudo' ]
format = 'image/jpeg'
crs = 'EPSG:4326'
rlayer = QgsRasterLayer(0, url, 'some_layer_name', 'wms', layers, styles, format, crs)
if not rlayer.isValid():
  print "Layer failed to load!"
```

## 2.3 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.

Добавление слоя в список:

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

При выходе слои уничтожаются автоматически, но если необходимо удалить слой явно используйте:

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

TODO: More about map layer registry?

# Работа с растровыми слоями

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

## 3.1 Информация о слое

Растровый слой состоит из одного или нескольких каналов и, соответственно, является одноканальным или многоканальным растром. Канал можно представить в виде матрицы значений. Обычно, цветные изображения (например, аэрофотоснимок) это растр состоящий из красного, зеленого и синего каналов. Одноканальные слои, как правило, представляют непрерывную (например, высота) или дискретную величину (например, использование земли). В некоторых случаях растровые слои поставляются с палитрой, и значения ячеек растра соответствуют цветам в палитре.

```
>>> rlayer.width(), rlayer.height()
(812, 301)
>>> rlayer.extent().toString()
PyQt4.QtCore.QString(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()
PyQt4.QtCore.QString(u'<p class="glossy">Driver:</p>...')
>>> rlayer.hasPyramids()
False
```

## 3.2 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 | Слой с “палитрой”, отрисовка с применением таблицы цветов |
| 4 | PalettedSingleBandGray | Слой с “палитрой”, отрисовка в оттенках серого |
| 5 | PalettedSingleBandPseudoColor | Слой с “палитрой”, отрисовка в псевдоцвете |
| 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. |

Узнать текущий стиль отображения можно так:

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

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

## 3.3 Одноканальные растры

По умолчанию отрисовка идет в оттенках серого. Чтобы изменить стиль отрисовки на псевдоцвет:

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

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

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

- линейный (INTERPOLATED): resulting color is linearly interpolated from the color map entries above and below the actual pixel value
- дискретный (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

To set an interpolated color ramp shader ranging from green to yellow color (for pixel values from 0 to 255):

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

Вернуться к стандартному отображению в оттенках серого можно так:

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

## 3.4 Многоканальные растры

По умолчанию, чтобы получить цветное изображение, QGIS ставит в соответствие трём первым каналам значения красного, зеленого и синего (это соответствует стилю отображения MultiBandColor). В некоторых случаях требуется переопределить эти настройки. Следующий код показывает как поменять местами красный (1) и зелёный (2) каналы:

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

Когда для визуализации слоя достаточно одного канала, можно выбрать отрисовку с использованием только одного канала — в оттенках серого или в псевдоцвете, см. предыдущий раздел:

```
>>> rlayer.setDrawingStyle(QgsRasterLayer.MultiBandSingleBandPseudoColor)
>>> rlayer.setGrayBandName(rlayer.bandName(1))
>>> rlayer.setColorShadingAlgorithm(QgsRasterLayer.PseudoColorShader)
>>> # далее идет установка шейдера
```

## 3.5 Обновление слоёв

Если символика слоя была изменена и необходимо сделать изменения видимыми пользователю, вызовите следующие методы:

```
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.

После изменения символики слоя (о том, как это сделать рассказано в разделах, посвящённых растровым и векторным слоям), может потребоваться обновить символику слоя в виджете списка слоёв (легенде). Ниже показано как это делается (iface это экземпляр QgisInterface):

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

## 3.6 Получение значений

Чтобы узнать значение каналов растрового слоя в определенной точке выполните:

```
res, ident = rlayer.identify(QgsPoint(15.30,40.98))
for (k,v) in ident.iteritems():
  print str(k),":",str(v)
```

Функция определения возвращает True/False (чтобы информировать об успехе или неудаче) и словарь — ключами выступают имена каналов, а в качестве значений — значения в заданной точке. И ключ, и значение являются экземплярами QString, поэтому, чтобы увидеть саму величину необходимо конвертировать их в строку Python (как это сделано в примере выше).

# Работа с векторными слоями

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

TODO: Editing, Layer vs. Data provider, ...

## 4.1 Обход объектов векторного слоя

Ниже показано как выполнить обход всех объектов векторного слоя. Чтобы читать объекты слоя необходимо инициализировать процесс получения при помощи select() а потом последовательно вызывать nextFeature()

```
provider = vlayer.dataProvider()

feat = QgsFeature()
allAttrs = provider.attributeIndexes()

# начинаем получение данных: для каждого объекта запрашиваем все атрибуты и геометрию
provider.select(allAttrs)

# получаем каждый объект вместе с геометрией и атрибутами
while provider.nextFeature(feat):

  # извлекаем геометрию
  geom = feat.geometry()
  print "Feature ID %d: " % feat.id() ,

  # отображаем информацю об объекте
  if geom.vectorType() == QGis.Point:
    x = geom.asPoint()
    print "Point: " + str(x)
  elif geom.vectorType() == QGis.Line:
    x = geom.asPolyline()
    print "Line: %d points" % len(x)
  elif geom.vectorType() == 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"

  # извлекаем атрибуты
  attrs = feat.attributeMap()

  # attrs это словарь: ключ = индекс поля, значение = QgsFeatureAttribute
  # показываем все атрибуты и их значения
  for (k,attr) in attrs.iteritems():
    print "%d: %s" % (k, attr.toString())
```

select() позволяет получать только необходимые данные. Функция может принимать 4 необязательных аргумента:

1. fetchAttributes Список атрибутов, которые нужно получать. По умолчанию: пустой список
2. rect Пространственный фильтр. Если передается пустой прямоугольник (QgsRectangle()), запрашиваются все объекты. По умолчанию: пустой прямоугольник
3. fetchGeometry Нужно ли запрашивать геометрию. По умолчанию: True
4. useIntersect При использовании пространственного фильтра определяет точность проверки на пересечение: будет ли использоваться точная проверка или достаточно проверки рамок. Это необходимо, например, при использовании функции определения или выбора. По умолчанию: False

Немного примеров:

```
# получение объектов с геометрией и только первыми двумя полями
provider.select([0,1])

# получение объектов, попадающих в заданный прямоугольник, запрашивается только геометрия
provider.select([], QgsRectangle(23.5, -10, 24.2, -7))

# получение объектов без геометрии, но со всеми атрибутами
allAtt = provider.attributeIndexes()
provider.select(allAtt, QgsRectangle(), False)
```

Для получения индекса поля по его имени используется функция провайдера данных fieldNameIndex():

```
fldDesc = provider.fieldNameIndex("DESCRIPTION")
if fldDesc == -1:
  print "Field not found!"
```

## 4.2 Редактирование векторных слоёв

Большинство провайдеров векторных данных поддерживает редактирование. Иногда они позволяют выполнять только некоторые операции редактирования. Узнать список доступных действий можно при помощи функции capabilities():

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

При использовании любого из следующих методов редактирования слоя, изменения будут применяться к соответствующему набору данных (файлу, базе данных и т.д) сразу же. Если необходимо произвести временные изменения, следующий раздел можно пропустить и перейти сразу к разделу, описывающему редактирование с использованием буфера изменений.

### 4.2.1 Добавление объектов

Создайте несколько экземпляров QgsFeature и передайте список этих объектов в метод addFeatures() провайдера. Провайдер вернет два значения: результат (true/false) и список добавленных объектов (из ID устанавливаются хранилищем данных):

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

### 4.2.2 Удаление объектов

Для удаления объектов достатовно передать список идентификаторов этих объектов:

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

### 4.2.3 Изменение объектов

Можно изменять как геометрию объекта так и его атрибуты. Следующий пример сначала изменяет значения атрибутов с индексами 0 и 1, а затем модифицирует геометрию объекта:

```
fid = 100   # ID объекта, который будет редактироваться

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

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

### 4.2.4 Добавление и удаление полей

Чтобы добавить поля (атрибуты), необходимо создать список с описанием полей. Для удаления необходимо предоставить список с индексами удаляемых полей.

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

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

## 4.3 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.

Определить находится ли слоё в режиме редактирования можно при помощи метода isEditing() — функции редактирования работают только при активном режиме редактирования. Применение операций редактирования показано ниже:

```
# добавление двух новых объектов (экземпляров QgsFeature)
layer.addFeatures([feat1,feat2])
# удаление объекта с заданным ID
layer.deleteFeature(fid)

# назначение новой геометрии (экземпляр QgsGeometry) объекту
layer.changeGeometry(fid, geometry)
# обновление атрибута с заданным индексом (int) заданным значением (QVariant)
layer.changeAttributeValue(fid, fieldIndex, value)

# добавление нового поля
layer.addAttribute(QgsField("mytext", QVariant.String))
# удаление поля
layer.deleteAttribute(fieldIndex)
```

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.

## 4.4 Использование пространственного индекса

TODO: Intro to spatial indexing

1. создание пространственного индекса — следующий код создаёт пустой индекс:

   ```
   index = QgsSpatialIndex()
   ```

2. добавление объектов к индексу — индекс принимает объект QgsFeature и добавляет его во внутреннюю структуру данных. Объект можно создать вручную или использовать полученные в результате предыдущих вызовов nextFeature()

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

3. после заполнения пространственного индекса значениями можно переходить к выполнению запросов:

   ```
   # возвращает массив идентификаторов пяти, ближайших к заданной точке, объектов
   nearest = index.nearestNeighbor(QgsPoint(25.4, 12.7), 5)

   # возвращает массив идентификаторов объектов, пересекающихся с заданным прямоугольником
   intersect = index.intersects(QgsRectangle(22.5, 15.3, 23.1, 17.2))
   ```

## 4.5 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:

- из экземпляра 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!"
```

Третий параметр задает конечную кодировку текста. Он требуется некоторым драйверам (в частности драйверу shape-файлов) для нормальной работы. В случае если вы не используете международные символы, специально заботиться о правильной кодировке не нужно. Четвертый параметр, который мы оставили пустым, задает целевую систему координат - если передан корректный экземпляр QgsCoordinateReferenceSystem, слой будет трансформирован в эту систему координат.

Узнать правильные названия драйверов можно на странице supported formats by OGR - в качестве имени драйвера используется значение колонки “Code”. При необходимости можно экспортировать только выделенные объекты, передать дополнительные параметры драйверу или запретить сохранение атрибутов - с полным синтаксисом можно ознакомиться в документации.

- из отдельных объектов:

```
# определяем поля для атрибутов объекта
fields = { 0 : QgsField("first", QVariant.Int),
           1 : QgsField("second", QVariant.String) }

# создаем экземпляр класса для записи векторных данных. Аргументы:
# 1. путь к новому файлу (если такой файл уже существует, возникнет ошибка)
# 2. кодировка атрибутивных данных
# 3. список полей
# 4. тип геометрии --- из перечислимого типа WKBTYPE
# 5. система координат слоя (экземпляр QgsCoordinateReferenceSystem) --- опционально
# 6. имя используемого драйвера
writer = QgsVectorFileWriter("my_shapes.shp", "CP1250", fields, QGis.WKBPoint, None, "ESRI Shapefile")

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

# добавляем объекты
fet = QgsFeature()
fet.setGeometry(QgsGeometry.fromPoint(QgsPoint(10,10)))
fet.addAttribute(0, QVariant(1))
fet.addAttribute(1, QVariant("text"))
writer.addFeature(fet)

# уничтожаем объект класса и сбрасываем изменения на диск (опционально)
del writer
```

## 4.6 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".

Начиная с QGIS 1.7 URI также может содержать описание системы координат, описание полей и включать пространственное индексирование. Используется следующий синтаксис:

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.

Ниже показан пример URI, содержащий все описанные выше параметры:

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

Следующий пример кода показывает процесс создания и наполнения memory провайдера:

```
# создается слой
vl = QgsVectorLayer("Point", "temporary_points", "memory")
pr = vl.dataProvider()

# добавляются поля
pr.addAttributes( [ QgsField("name", QVariant.String),
                    QgsField("age",  QVariant.Int),
                    QgsField("size", QVariant.Double) ] )

# добавляется объект
fet = QgsFeature()
fet.setGeometry( QgsGeometry.fromPoint(QgsPoint(10,10)) )
fet.setAttributeMap( { 0 : QVariant("Johny"),
                       1 : QVariant(20),
                       2 : QVariant(0.3) } )
pr.addFeatures( [ fet ] )

# после добавления новых объектов обновляем охват слоя
# т.к. изменение охвата в провайдере не распространяется на слой
vl.updateExtents()
```

И, наконец, проверим что всё прошло успешно:

```
# отображаем информацию о слое
print "fields:", pr.fieldCount()
print "features:", pr.featureCount()
e = pr.extent()
print "extent:", e.xMin(),e.yMin(),e.xMax(),e.yMax()

# проходим по всем объектам
f = QgsFeature()
pr.select()
while pr.nextFeature(f):
  print "F:",f.id(), f.attributeMap(), f.geometry().asPoint()
```

## 4.7 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.

В QGIS v1.4 был введен новый механизм отрисовки векторных слоев, призванный устранить ограничения оригинальной реализации. Мы называем его новой символикой или symbology-ng (new generation — новое поколение), оригинальную реализацию также называют старой символикой. Каждый векторный слой использует либо новую либо старую символику, но нельзя использовать обе одновременно, или ни одну из них. Это не глобальная настройка для всех слоёв, то есть некоторые слои могут использовать новую символику, в то время как другие продолжают использовать старую. В настройках QGIS пользователь может указать какую символику необходимо использовать по умолчанию. Старая символика будет сохранена в будущих выпусках QGIS 1.x для совместимости, но мы планируем отказаться от неё в QGIS v2.0.

Узнать какая реализация используется можно так:

```
if layer.isUsingRendererV2():
  # новая символика --- подкласс класса QgsFeatureRendererV2
  rendererV2 = layer.rendererV2()
else:
  # старая символика --- подкласс класса QgsRenderer
  renderer = layer.renderer()
```

Примечание: если требуется поддержка старых версий (например, QGIS $< 1.4$), вначале необходимо проверить существует ли метод isUsingRendererV2() — если его нет, доступна только старая символика:

```
if not hasattr(layer, 'isUsingRendererV2'):
  print "You have an old version of QGIS"
```

В первую очередь и более подробно мы рассмотрим новую символику, так как она обладает большими возможностями и предоставляет больше настроек.

### 4.7.1 Новая символика

Теперь, имея ссылку на рендер из предыдущего раздела, изучим его поближе:

```
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 |

Кроме того, могут быть доступны пользовательские рендеры, поэтому не стоит предполагать, что присутствуют только вышеназванные типы. Узнать список доступных рендеров можно обратившись к синглтону QgsRendererV2Registry.

Существует возможность получить дамп содержимого рендера в текстовом виде, это может быть полезно при отладке:

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

### Рендер обычным знаком

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).

### Рендер уникальными значениями

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

А так получаем список значений:

```
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.

### Рендер градуированным знаком

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.

Получить информацию о диапазонах, используемых рендером:

```
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.

Если вы хотите создать свой рендер категориями, можете воспользоваться следующим фрагментом кода в качестве отправной точки. Пример ниже создает простое разделение объектов на два класса:

```
from qgis.core import (QgsVectorLayer, QgsMapLayerRegistry,
                       QgsGraduatedSymbolRendererV2,
                       QgsSymbolV2, QgsRendererRangeV2)

myVectorLayer = QgsVectorLayer(myVectorPath, myName, 'ogr')
myTargetField = myStyle['target_field']
myRangeList = []
myOpacity = 1
# создаем первый символ и диапазон...
myMin = 0.0
myMax = 50.0
myLabel = 'Group 1'
myColour = QtGui.QColor('#ffee00')
mySymbol1 = QgsSymbolV2.defaultSymbol(
    myVectorLayer.geometryType())
mySymbol.setColor(myColour)
mySymbol.setAlpha(myOpacity)
myRange1 = QgsRendererRangeV2(
        myMin,
        myMax,
        mySymbol1,
        myLabel)
myRangeList.append(myRange1)
# теперь создаем другой символ и диапазоне...
myMin = 50.1
myMax = 100
myLabel = 'Group 2'
myColour = QtGui.QColor('#00eeff')
mySymbol2 = QgsSymbolV2.defaultSymbol(
    myVectorLayer.geometryType())
mySymbol.setColor(myColour)
mySymbol.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)
```

### Работа с символами

Символы представлены базовым классом QgsSymbolV2 и тремя классами наследниками:

- QgsMarkerSymbolV2 - для точечных объектов
- QgsLineSymbolV2 - для линейных объектов
- QgsFillSymbolV2 - для полигональных объектов

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.

Получив экземпляр символа (например, от рендера), можно заняться его изучением: метод type() расскажет является ли этот символ маркером, линией или заливкой. Метод dump() вернет краткое описание символа. А получить список слоёв символа можно так:

```
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.

### Работа со слоями символа

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.

Получить полный список символьных слоёв, которые можно использовать в заданном символьном слое, можно так:

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

Результат:

```
EllipseMarker
FontMarker
SimpleMarker
SvgMarker
VectorField
```

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.

### Создание пользовательских символьных слоёв

Представьте, что вам необходимо настроить процесс отрисовки своих данных. Для этого можно создать свой собственный класс символьного слоя, который будет рисовать объекты именно так, как вам нужно. Вот пример маркера, рисующего красные окружности заданного радиуса:

```
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):
    # Отрисовка зависит от того выделен символ или нет (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 трансформированы в выходные координаты.

Для полилиний 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.

Хорошей практикой является создание интерфейса для управления атрибутами символьного слоя, что позволяет пользователям настраивать внешний вид: в случае нашего примера, можно предоставить пользователю возможность менять радиус окружности. Реализовать это можно так:

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

    self.layer = None

    # создаем простой интерфейс
    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.

Сначала нужно создать метаданные символьного слоя:

```
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.

### Создание пользовательских рендеров

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.

Следующий код демонстрирует простой пользовательский рендер, который создает два маркера и случаным образом выбирает один из них при отрисовке каждого объекта:

```
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.

Как и в случае символьных слоёв, рендер может иметь интерфейс для настройки параметров. Он наследуется от класса QgsRendererV2Widget. Следующий код создает кнопку, позволяющую пользователю изменять один из символов:

```
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
    # создание интерфейса
    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.

Осталось немного: метаданные рендера и его регистрация в реестре, иначе загрузить слои с этим рендером не получится, а пользователь не увидит его в списке доступных рендеров. Закончим наш пример с RandomRenderer:

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

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

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

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.

Назначить иконку рендеру можно передав её в конструктор QgsRendererV2AbstractMetadata в качестве третьего (необязательного) аргумента — конструктор базового класса в функции __init__ класса RandomRendererMetadata примет вид:

```
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).

### Further Topics

TODO:

- creating/modifying symbols
- working with style (QgsStyleV2)
- working with color ramps (QgsVectorColorRampV2)
- rule-based renderer
- exploring symbol layer and renderer registries

### 4.7.2 Старая символика

Знак определяет цвет, размер и другие свойства объекта. Рендер, ассоциированный со слоем, решает какой знак будет использован для определённого объекта. Всего доступно четыре рендера:

- обычный знак (QgsSingleSymbolRenderer) — все объекты отображаются одним и тем же знаком.
- уникальное значение (QgsUniqueValueRenderer) — знак для каждого объекта выбирается на основании значения атрибута.
- градуированный знак (QgsGraduatedSymbolRenderer) — знак применяется к группе (классу) объектов, разбиение на классы выполняется по числовому полю
- непрерывное значение (QgsContinuousSymbolRenderer)

Создать точечный знак можно так:

```
sym = QgsSymbol(QGis.Point)
sym.setColor(Qt.black)
sym.setFillColor(Qt.green)
sym.setFillStyle(Qt.SolidPattern)
sym.setLineWidth(0.3)
sym.setPointSize(3)
sym.setNamedPointSymbol("hard:triangle")
```

Метод setNamedPointSymbol() определяет фигуру, которая будет использоваться для знака. Существует два класса: встроенные символы (начинается с hard:) и SVG символы (начинаются с svg:). Доступны следующие встроенные символы: circle, rectangle, diamond, pentagon, cross, cross2, triangle, equilateral_triangle, star, regular_star, arrow.

Для создания SVG знака выполните:

```
sym = QgsSymbol(QGis.Point)
sym.setNamedPointSymbol("svg:Star1.svg")
sym.setPointSize(3)
```

SVG символы не поддерживают установку цвета, заливки и стиля линии.

Создание линейного знака:

TODO

Создание площадного знака:

TODO

Создание рендера обычным знаком:

```
sr = QgsSingleSymbolRenderer(QGis.Point)
sr.addSymbol(sym)
```

Назначение рендера слою:

```
layer.setRenderer(sr)
```

Рендер уникальным значением:

TODO

Рендер градуированным значением:

```
# задаем поле классификации и желаемое количество классов
fieldName = "My_Field"
numberOfClasses = 5

# получаем номер поля по его имени
fieldIndex = layer.fieldNameIndex(fieldName)

# создаем ренедер, которые позже будет назначен слою
renderer = QgsGraduatedSymbolRenderer( layer.geometryType() )

# настраиваем режим рендера, можно выбрать один из EqualInterval/Quantile/Empty
renderer.setMode( QgsGraduatedSymbolRenderer.EqualInterval )

# задаем классы (нижнюю и верхнюю границы и подпись для каждого класса)
provider = layer.dataProvider()
minimum = provider.minimumValue( fieldIndex ).toDouble()[ 0 ]
maximum = provider.maximumValue( fieldIndex ).toDouble()[ 0 ]

for i in range( numberOfClasses ):
    # строку форматирования надо изменить в зависимости  от типа данных (целые или с плавающей точкой)
    lower = ('%.*f' % (2, minimum + ( maximum - minimum ) / numberOfClasses * i ) )
    upper = ('%.*f' % (2, minimum + ( maximum - minimum ) / numberOfClasses * ( i + 1 ) ) )
    label = "%s - %s" % (lower, upper)
    color = QColor(255*i/numberOfClasses, 0, 255-255*i/numberOfClasses)
    sym = QgsSymbol( layer.geometryType(), lower, upper, label, color )
    renderer.addSymbol( sym )

# устанавливаем поле классификации и назначаем рендер слою
renderer.setClassificationField( fieldIndex )

layer.setRenderer( renderer )
```

# Обработка геометрии

Точки, линии, полигоны, являющиеся пространственными объектами обычно называют геометрией. В QGIS все они представлены классом QgsGeometry. Посмотреть на все существующие типы геометрий можно на странице обсуждения JTS.

Иногда одна геометрия на самом деле является множеством простых (single-part) объектов. Такую геометрию называют составной (multi-part). Если составная геометрия состоит из простых объектов одного типа, то ее называют мульти-точкой, мульти-линией или мульти-полигоном. Например, страна, состоящая из нескольких островов может быть представлена как мульти-полигон.

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.

## 5.1 Создание геометрий

Создать геометрию можно несколькими способами:

- по координатам:

  ```
  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.

- из описания в формате WKT (well-known text):

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

- из описания в формате WKB (well-known binary):

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

## 5.2 Доступ к геометрии

Прежде всего необходимо определить тип геометрии, сделать это можно вызвав метод wkbType(), который вернет значение из перечислимого типа QGis.WkbType:

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

Также можно воспользоваться методом type(), который возвращает значение из перечислимого типа QGis.GeometryType. Вспомогательная функция isMultipart() поможет определить является ли геометрия составной или нет.

Для извлечения информации из геометрии существуют фукции доступа для каждого вида объектов. Ниже показано как ими пользоваться:

```
>>> 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().

## 5.3 Геометрические предикаты и операции

QGIS использует библиотеку GEOS для выполнения различных действий над геометриями, таких как геометрические предикаты (contains(), intersects(), ...) и операции (union(), difference(), ...)

TODO:

- area(), length(), distance()
- transform()
- available predicates and set operations

# Работа с проекциями

## 6.1 Системы координат

Системы координат (Coordinate reference system, CRS) инкапсулируются классом QgsCoordinateReferenceSystem. Создать экземпляр этого класса можно одним из способов:

- задать CRS по её ID:

  ```
  # WGS84 имеет PostGIS SRID 4326
  crs = QgsCoordinateReferenceSystem(4326, QgsCoordinateReferenceSystem.PostgisCrsId)
  ```

  QGIS использует три разных идентификатора для каждой системы координат:

  - PostgisCrsId — идентификатор, используемый в базах PostGIS.
  - InternalCrsId — внутренний идентификатор, используемый в базе QGISe.
  - EpsgCrsId — идентификатор, назначенный EPSG

  По умолчанию используется PostGIS SRID, если иное не определено вторым параметром.

- задать CRS по её представлению в формате WKT (well-known text):

  ```
  wkt = 'GEOGCS["WGS84", DATUM["WGS84", SPHEROID["WGS84", 6378137.0, 298.257223563]],\
      PRIMEM["Greenwich", 0.0], UNIT["degree",0.01745329251994295],\
      AXIS["Longitude",EAST], AXIS["Latitude",NORTH]]'
  crs = QgsCoordinateReferenceSystem(wkt)
  ```

- создать недействительную CRS, а затем использовать одну из функций create*() для её инициализации. В примере ниже для инициализации проекции используется строка в формате Proj4:

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

Желательно проверить успешность создания (т.е выполнить поиск в базе данных) системы координат: isValid() должна вернуть True.

Заметим, что для инициализации систем координат QGIS требуется выполнить поиск необходимых значений в своей внутренней базе данных srs.db. Поэтому, если создаётся самостоятельное приложение, необходимо правильно настроить пути при помощи QgsApplication.setPrefixPath(), иначе не удастся найти базу данных. В случае создания расширения или при работе в консоли Python QGIS беспокоиться не о чем: всё уже настроено.

Получение информации о системе координат:

```
print "QGIS CRS ID:", crs.srsid()
print "PostGIS SRID:", crs.srid()
print "EPSG ID:", crs.epsg()
print "Description:", crs.description()
print "Projection Acronym:", crs.projectionAcronym()
print "Ellipsoid Acronym:", crs.ellipsoidAcronym()
print "Proj4 String:", crs.proj4String()
# определяем тип системы координат: географическая или спроецированная
print "Is geographic:", crs.geographicFlag()
# единицы карты, используемые в этой CRS (определены в перечислимом типе QGis::units)
print "Map units:", crs.mapUnits()
```

## 6.2 Проекции

Для преобразования между разными системами координат используется класс QgsCoordinateTransform. Наиболее простой способ использования — создать объекты для исходной и целевой систем координат и инициализировать ими экземпляр класса QgsCoordinateTransform. После чего можно выполнять преобразование, вызывая функцию transform(). По умолчанию выполняется прямое преобразование, но можно осуществлять и обратное:

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

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

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

# 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.

Карта реализуется классом QgsMapCanvas модуля qgis.gui. В основе реализации лежит Qt Graphics View framework. Фреймворк предоставляет пользователю поверхность для рисования и объект для отображения пользовательских элементов, а также даёт возможность взаимодействовать с ними. Предполагается, что читатель достаточно знаком с Qt чтобы разобраться в основных понятиях сцены, вида и элементов. Если это не так, пожалуйста, ознакомьтесь с описанием фреймворка.

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.

## 7.1 Встраивание карты

Так как карта это такой же элемент интерфейса, как и любой другой виджет Qt, её использование, создание и отображение весьма просто:

```
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.

По умолчанию, фон карты чёрный, сглаживание при отрисовке отключено. Чтобы установит цвет фона в белый и активировать сглаживание выполните:

```
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.)

Теперь можно добавить несколько слоёв. Сначала слой необходимо открыть и добавить его к списку слоёв карты. Затем нужно установить охват и добавить слои к карте:

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

# добавляем слой к списку
QgsMapLayerRegistry.instance().addMapLayer(layer)

# устанавливаем охват карты равный охвату слоя
canvas.setExtent(layer.extent())

# добавляем слои на карту
canvas.setLayerSet( [ QgsMapCanvasLayer(layer) ] )
```

После выполнения этих команд на карте должен отобразиться загруженный слой.

## 7.2 Использование инструментов карты

Следующий пример показывает как создать окно с картой и основными инструментами для панорамирования и масштабирования карты. Для каждого инструмента создаётся свое действие: за панорамирование отвечает QgsMapToolPan, за увеличение и уменьшение масштаба — QgsMapToolZoom. Действия настроены на работу в режиме переключателя и позже будут связанны с инструментами, что позволит автоматически отслеживать переключение между ними. Когда инструмент карты активируется, его действие помечается как активное, а действие, связанное с предыдущим инструментом, — как неактивное. За активацию инструментов карты отвечает метод setMapTool().

```
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)

    # создаем инструменты карты
    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)
```

Этот код можно сохранить в файл, например, mywnd.py и попробовать выполнить в Консоли Python QGIS. Код ниже показывает как поместить текущий выделенный слой на созданную только что карту:

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

Перед этим необходимо убедиться, что файл mywnd.py находится в каталоге где Python ищет мо-

дули (sys.path). Если это не так, просто добавьте его: sys.path.insert(0, '/my/path') — иначе импорт завершится с ошибкой, из-за того, что модуль не найден.

## 7.3 Резиновые полосы и маркеры вершин

Для отображения дополнительных данных поверх карты используются элементы карты. Можно как создавать свои собственные элементы карты (рассматривается дальше), так и использовать существующие классы: QgsRubberBand для рисования полигонов или полилиний, и QgsVertexMarker для рисования точек. Оба этих класса работают в координатах карты, поэтому фигуры автоматически перемещаются/масштабируются при панорамировании и масштабировании карты

Показать полилинию можно так:

```
r = QgsRubberBand(canvas, False)  # False = не полигон
points = [ QgsPoint(-1,-1), QgsPoint(0,1), QgsPoint(1,-1) ]
r.setToGeometry(QgsGeometry.fromPolyline(points), None)
```

Отобразить полигон:

```
r = QgsRubberBand(canvas, True)  # True = полигон
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.

Резиновые полосы можно настраивать, а именно менять их цвет и толщину:

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

Элементы карты связанны с графической сценой карты. Их можно скрыть (а потом снова отобразить) вызывая функции func:hide и show(). Для полного удаления элемента необходимо удалить его из графической сцены:

```
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)

Резиновые полосы можно использовать и для рисования точек, но для этих целей существует специальный класс QgsVertexMarker (QgsRubberBand может нарисовать только прямоугольник вокруг заданной точки).

Вот так можно создать маркер вершины:

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

Следующий фрагмент кода показывает как создается красный крестик в точке [0,0]. Можно настроить тип значка, его размер, цвет и толщину пера:

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

Для временно скрытия и последующего отображения маркеров используется тот же подход, что и для резиновых полос.

## 7.4 Создание собственных инструментов карты

TODO: how to create a map tool

## 7.5 Создание собственных элементов карты

TODO: how to create a map canvas item

# 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.

## 8.1 Простая отрисовка

Отрисовка нескольких слоёв с помощью QgsMapRenderer — создаётся объект для рисования (QImage, QPainter и др.), задаётся набор слоёв, охват, размер результата и выполняется рендеринг:

```
# создаём изображение
img = QImage(QSize(800,600), QImage.Format_ARGB32_Premultiplied)

# устанавливаем цвет фона
color = QColor(255,255,255)
img.fill(color.rgb())

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

render = QgsMapRender()

# задаём набор слоёв
lst = [ layer.getLayerID() ]  # добавляем ID необходимых слоёв
render.setLayerSet(lst)

# устанавливаем охват
rect = QgsRect(render.fullExtent())
rect.scale(1.1)
render.setExtent(rect)

# устанавливаем размер результата
render.setOutputSize(img.size(), img.logicalDpiX())

# выполняем отрисовку
render.render(p)
```

```
p.end()

# сохраняем изображение
img.save("render.png","png")
```

## 8.2 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.

Компоновщик состоит из множества классов. Все они являются частью библиотеки ядра. В QGIS присутствует удобный интерфейс пользователя для расстановки всех элементов, но он пока ещё не доступен через библиотеку графического интерфейса. Если вы не знакомы с Qt Graphics View framework, рекомендуем изучить документацию сейчас, потому что компоновщик основан на этом фреймворке.

Основным классом компоновщика является QgsComposition, который в свою очередь основан на QGraphicsScene. Создадим экземпляр этого класса:

```
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:

- карта — этот элемент определяет положение карты. Так можно создать карту и растянуть её на весь лист:

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

- подпись — позволяет отображать подписи. Можно изменять шрифт, цвет, выравнивание и поля

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

- легенда

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

- масштабная линейка

```
item = QgsComposerScaleBar(c)
item.setStyle('Numeric') # при желании стиль можно изменить
item.setComposerMap(composerMap)
item.applyDefaultSize()
c.addItem(item)
```

- стрелка севера
- изображение
- фигура
- таблица

По умолчанию только что созданные элементы компоновки имеют нулевое положение (левый верхний угол страницы) и нулевой размер. Положение и размер всегда задаюся в миллиметрах:

```
# расположить подпись на расстоянии 1 см от верхнего края и 2 см от левого края страницы
composerLabel.setItemPosition(20,10)
# установить размер и положение метки (ширина 10 см, высота 3 см)
composerLabel.setItemPosition(20,10, 100, 30)
```

Вокруг каждого элемента по умолчанию рисуется рамка. Убрать её можно так:

```
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.

По умолчанию для вывода используется лист формата A4 и разрешение 300 DPI. При необходимости эти настройки можно изменить. Размер бумаги задаётся в миллиметрах:

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

### 8.2.1 Вывод в растровое изображение

Следующий фрагмент кода показывает как вывести компоновку в растровое изображение:

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

# создаём выходное изображение и инициализируем его
image = QImage(QSize(width, height), QImage.Format_ARGB32)
image.setDotsPerMeterX(dpmm * 1000)
image.setDotsPerMeterY(dpmm * 1000)
image.fill(0)

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

```
imagePainter.end()

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

### 8.2.2 Вывод в формате PDF

Следующий пример иллюстрирует вывод компоновки в файл формата PDF:

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

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

# 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
- срока — 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

Поддерживаются предикаты:

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

Примечание по совместимости: математические и геометрические функции, функции преобразования типа и оператор возведения в степень ^ доступны начиная с QGIS 1.4.

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

## 9.1 Разбор выражений

TODO: parsing, error handling

```
>>> s = QgsSearchString()
>>> s.setString("1 + 1 = 2")
True
>>> s.setString("1 + 1 =")
False
>>> s.parserErrorMsg()
PyQt4.QtCore.QString(u'syntax error, unexpected $end')
```

TODO: working with the tree, evaluation as a predicate, as a function, error handling

## 9.2 Вычисление выражений

```
st = ss.tree()
if not st:
  raise ValueError, "empty expression was used"

print st.makeSearchString()

res = st.checkAgainst(fields, feature.attributeMap())

res, value = st.getValue(st, fields, feature.attributeMap(), feature.geometry())

print st.errorMsg()
```

# Измерения

Для вычисления расстояний или площадей используется класс QgsDistanceArea. Если перепроецирование выключено, расчёт будет выполняться на плоскости, в противном случае — на эллипсоиде. Когда эллипсоид не задан, для расчётов используются параметры WGS84.

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

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

TODO: area, planar vs. ellipsoid

# Чтение и сохранение настроек

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

Эти параметры можно сохранять и получать при помощи Qt и QGIS API. Для каждого параметра необходимо выбрать ключ, который будет использоваться для доступа к переменной — так, для предпочитаемого цвета можно использовать ключ “favorite_color” или любую другую подходящую по смыслу строку. Рекомендуется придерживаться некоторой системы в именовании ключей.

Необходимо различать следующие типы настроек:

- глобальные настройки — они свои для каждого пользователя компьютера. QGIS сама хранит множество глобальных настроек, например, размер главного окна или порог прилипания. Эта функциональность обеспечивается Qt, точнее входящим в ее состав классом QSettings. По умолчанию, этот класс хранит настройки в “родном” для системы виде — реестр (для Windows), файл .plist (на Mac OS X) или .ini файл (в Unix). Описание QSettings documentation достаточно обширное, поэтому ограничимся простым примером:

  ```
  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").toString()
    myint  = s.value("myplugin/myint", 123).toInt()[0]
    myreal = s.value("myplugin/myreal", 2.71).toDouble()[0]
  ```

  Qt использует экземпляры QVariant для значений переменных в методах setValue() и value(). Значения переменных автоматически конвертируются из переменных Python в экземпляры QVariant, однако обратное преобразование из QVariant в Python таковым не является: поэтому используются методы to*(). Также обратите внимание на следующие моменты:

  - второй параметр метода value() опциональный и содержит значение по умолчанию, на случай если по каким-либо причинам считать значение не удастся
  - toString() возвращает экземпляр QString, а не строку Python
  - toInt() и toDouble() возвращают кортеж (value, ok) — второй элемент имеет значение True если преобразование из QVariant в число прошло успешно — в примере мы игнорируем этот

флаг и получаем только значение.

- настройки проекта разные для каждого проекта и поэтому они связаны с файлом проекта. Примером могут служить цвет фона карты и используемая система координат (CRS) — в одном проекте может быть белый фон и WGS84, а в другом желтый фон и проекция UTM. Пример использования ниже:

  ```
  proj = QgsProject.instance()

  # сохранение значение
  proj.writeEntry("myplugin", "mytext", "hello world")
  proj.writeEntry("myplugin", "myint", 10)

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

  Возможно, в дальнейшем класс QgsProject будет обновлен, с тем чтобы предоставлять API, похожее на класс QSettings. Из-за некоторых ограничений привязок Python сохранение чисел с плавающей запятой не возможно.

- настройки слоя — эти настройки относятся к отдельному экземпляру слоя карты. Они не связаны с определеным источником данных слоя, поэтому если создано два экземпляра слоя из одного shape-файла, они не будут использовать эти настройки совместно. Настройки хранятся в файле проекта, поэтому при открытии проекта настройки слоя будут восстановлены. Этот функционал добавлен в QGIS v1.4. API похоже на используемое в QSettings — для чтения и записи настроек используются экземпляры QVariant:

  ```
  # сохранить значение
  layer.setCustomProperty("mytext", "hello world")

  # прочитать значение
  mytext = layer.customProperty("mytext", "default text").toString()
  ```

TODO: Keys for settings that can be shared among plugins

# 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, ...

## 12.1 Наследование QgsPluginLayer

Ниже показан пример минимальной реализации QgsPluginLayer. Это фрагмент 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
```

При необходимости можно добавить методы для чтения и записи информации в файл проекта:

```
def readXml(self, node):

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

Для загрузки проекта, содержащего такой слой, требуется наличие класса:

```
class WatermarkPluginLayerType(QgsPluginLayerType):

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

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

Кроме того, можно добавить код для отображения дополнительной информации в окне свойств слоя:

def showLayerProperties(self, layer):

# Библиотека сетевого анализа

Начиная с ee19294562 (QGIS >= 1.8) в ядре QGIS появилась библиотека сетевого анализа, которая:

- может создавать математический граф из географических данных (линейных векторных слоёв)
- реализует базовые методы теории графов (в настоящее время только метод Дейкстры)

Библиотека сетевого анализа является результатом экспорта основных функций модуля RoadGraph, и теперь этот функционал можно использовать в своих расширениях, а также из Консоли Python QGIS.

## 13.1 Применение

Типичный алгоритм использования библиотеки описывается следующими шагами:

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

## 13.2 Получение графа

Первое, что нужно сделать — это подготовить исходные данные, т.е. преобразовать векторный слой в граф. Все дальнейшие действия будут выполняться именно с этим графом.

В качестве источника графа может выступать любой линейный векторный слой. Узлы линий образуют множество вершин графа. В качестве ребер графа выступают отрезки линий векторного слоя. Узлы, имеющие одинаковые координаты, считаются одной и той же вершиной графа. Таким образом, две линии, имеющие общий узел, оказываются связанными между собой.

В дополнение к этому, при построении графа можно «привязать» к векторному слою любое количество дополнительных точек. Для каждой дополнительной точки будет найдено соответствие — либо ближайшая вершина графа, либо ближайшее ребро. В последнем случае ребро будет разбито на две части и будет добавлена новая общая вершина.

В качестве свойств ребер графа могут быть использованы атрибуты векторного слоя и протяженность (длина) ребра.

Реализация построения графа из векторного слоя использует шаблон программирования строитель. а построение графа дорог отвечает так называемый Director. В настоящее время бибилотека располагает только одним директором: QgsLineVectorLayerDirector. Директор задает основные настройки, которые будут использоваться при построении графа по линейному векторному слою, и «руками» строителя выполняет построение графа. В настоящее время, как и в случае с директором, реализован только один строитель: QgsGraphBuilder, создающий графы типа QgsGraph. При желании можно реализовать строителя, который будет строить граф,совместимый с такими библиотеками как BGL или NetworkX.

Для вычисления свойств ребер используется шаблон проектирования стратегия. Пока в библиотеке реализована только одна стратегия, учитывающая длину маршрута QgsDistanceArcProperter. При необходимости, можно создать свою стратегию, которая будет учитывать нужные параметры. Например, в модуле Road graph используется стратегия, вычисляющая время движения по ребру графа на основании длины ребра и поля скорости.

Рассмотрим процесс создание графа более подробно. Чтобы получить доступ к функциям библиотеки сетевого анализа необходимо импортировать модуль networkanalysis:

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

Теперь нужно создать директора:

```
# не использовать информацию о направлении движения из атрибутов слоя,
# все дороги трактуются как двустронние
director = QgsLineVectorLayerDirector( vLayer, -1, '', '', '', 3 )

# информация о направлении движения находится в поле с индексом 5.
# Односторонние дороги с прямым направлением движения имееют значение
# атрибута "yes", односторонние дороги с обратным направлением — "1",
# и соответственно двусторонние дороги — "no". По умолчанию дороги
# считаются двусторонними. Такая схема подходит для использования
# с данными OpenStreetMap
director = QgsLineVectorLayerDirector( vLayer, 5, 'yes', '1', 'no', 3 )
```

В конструктор директора передается линейный векторный слой, по которому будет строиться граф, а также информация о характере движения по каждому сегменту дороги (разрешенное направление, одностороннее или двустороннее движение). Рассмотрим эти параметры:

- vl — векторный слой, по которому будет строиться граф
- directionFieldId — индекс поля атрибутивной таблицы, которое содержит информацию о направлении движения. -1 не использовать эту информацию
- directDirectionValue — значение поля, соответствующее прямому направлению движения (т.е. движению в порядке создания точек линии, от первой к последней)
- reverseDirectionValue — значение поля, соответствующее обратному направлению движения (от последней точки к первой)
- bothDirectionValue — значение поля, соответствующее двустроннему движению (т.е. допускается движение как от первой точки к последней, так и в обратном направлении)
- defaultDirection — направление движения по умолчанию. Будет использоваться для тех участков дорог, у которых значение поля directionFieldId не задано или не совпадает ни с одним из вышеперечисленных

Следующим шагом необходимо создать стратегию назначения свойств ребрам графа:

```
properter = QgsDistanceArcProperter()
```

Сообщаем директору об используемой стратегии. Один директор может использовать несколько стратегий

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

Теперь создаем строителя, который собственно и будет строить граф заданного типа. Конструктор :class:QgsGraphBuilder принимает следующие параметры:

- 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”

```
# задана только используемая СК, все остальные параметры по умолчанию
builder = QgsGraphBuilder( myCRS )
```

Также можно задать одну или несколько точек, которые будет использоваться при анализе. Например так:

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

Затем строим граф и «привязываем» к нему точки:

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

Построение графа может занять некоторое время (зависит от количества объектов в слое и размера самого слоя). В tiedPoints записываются координаты «привязанных» точек. После построения мы получим граф, пригодный для анализа:

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

Теперь можно получить индексы наших точек:

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

## 13.3 Анализ графа

В основе сетевого анализа лежат задача связности вершин графа и задача поиска кратчайших путей. Для решения этих задач в библиотеке network-analysis реализован алгоритм Дейкстры.

Алгоритм Дейкстры находит оптимальный маршрут от одной из вершин графа до всех остальных и значение оптимизируемого параметра. Хорошим способом представления результата выполнения алгоритма Дейкстры является дерево кратчайших путей.

Дерево кратчайших путей — это ориентированный взвешенный граф (точнее дерево) обладающий следующими свойствами:

- только одна вершина не имеет входящих в нее ребер — корень дерева
- все остальные вершины имеют только одно входящее в них ребро
- Если вершина B достижима из вершины A, то путь, соединяющий их, единственный и он же кратчайший (оптимальный) на исходном графе

Дерево кратчайших путей можно получить вызывая методы shortestTree() и dijkstra() класса QgsGraphAnalyzer. Рекомендуется пользоваться именно методом dijkstra(), т.к. он работает быстрее и, в общем случае, эффективнее расходует память.

Метод shortestTree() может быть полезен в тех случаях когда необходимо совершить обход дерева кратчайших путей. Он создает новый объект (всегда QgsGraph) и принимает три аргумента:

- 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 )
```

Вот так выглядит простейший способ отобразить дерево кратчайших путей с использованием графа, полученного в результате вызова метода shortestTree() (только замените координаты начальной точки на свои, а также выделите слой дорог в списке слоёв карты). Внимание: код создает огромное количество объектов QgsRubberBand используйте его только в качестве примера и для очень маленьких слоев.

```
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
```

То же самое, но с использованием метода 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() )
```

### 13.3.1 Нахождение кратчайших путей

Для получения оптимального маршрута между двумя произвольными точками используется следующий подход. Обе точки (начальная A и конечная B) «привязываются» к графу на этапе построения, затем при помощи метода shortestTree() или dijkstra() находится дерево кратчайших маршрутов с корнем в начальной точке A. В этом же дереве находим конечную точку B и начинаем спуск по дереву от точки B к точке A. В общем виде алгоритм можно записать так:

```
присвоить T = B
пока T != A
    добавить точку T в маршрут
    найти ребро, входящее в точку T
    найти точку TT, из которой это ребро выходит
    присвоить T = TT
добавить точку A к маршруту
```

На этом построение маршрута закончено. Мы получили инвертированный список вершин (т.е. вершины идут в обратном порядке, от конечной точки к начальной), которые будут посещены при движении по кратчайшему маршруту.

Вот работающий пример поиска кратчайшего маршрута для Консоли Python QGIS (только замените координаты начальной и конечной точки на свои, а также выделите слой дорог в списке слоёв карты) с использованием метода 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)
```

А вот пример с использованием метода dikstra():

```
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)
```

### 13.3.2 Нахождение областей доступности

Назовем областью доступности вершины графа A такое подмножество вершин графа, доступных из вершины A, что стоимость оптимального пути от A до элементов этого множества не превосходит некоторого заданного значения.

Более наглядно это определение можно объяснить на следующем примере: «Есть пожарное депо. В какую часть города сможет попасть пожарная машина за 5 минут, 10 минут, 15 минут?». Ответом на этот вопрос и являются области доступности пожарного депо.

Поиск областей доступности легко реализовать при помощи метода dijksta() класса QgsGraphAnalyzer. Достаточно сравнить элементы возвращаемого значения с заданным параметром. Если величина cost[ i ] меньше заданного параметра или равна ему, тогда i-я вершина графа принадлежит множеству доступности, в противном случае — не принадлежит.

Не столь очевидным является нахождение границ доступности. Нижняя граница доступности — множество вершин которые еще можно достигнуть, а верхняя граница — множество вершин которых уже нельзя достигнуть. На самом деле все просто: граница доступности проходит по таким ребрам дерева кратчайших путей, для которых вершина-источник ребра доступна, а вершина-цель недоступна.

Вот пример:

```
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 ) )
```

# 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.

Расширения на Python перечисляются в Менеджере модулей QGIS наравне с расширениями на C++. Поиск расширений выполняется в следующих каталогах:

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

В Windows домашний каталог (обозначенный выше как ~) обычно выглядит как C:\Documents and Settings\(user). Вложенные каталоги в этих папках рассматриваются как пакеты Python, которые можно загружать в QGIS как расширения.

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. Создание файлов: Подробнее этот шаг описан ниже. Точка входа (__init.py__). Тело расширения (plugin.py). Форма QT-Designer (form.ui), со своим resources.qrc.
3. Реализация: Пишем код в plugin.py
4. Тестирование: Закройте и снова откройте QGIS, загрузите своё расширение. Проверьте, что всё работает как надо.
5. Публикация: опубликуйте своё расширение в репозитории QGIS или настройте свой собственный репозиторий в качестве “арсенала” личного “ГИС вооружения”

## 14.1 Разработка расширения

С момента введения поддержки Python в QGIS появилось множество расширений — на странице Plugin Repositories можно найти некоторые из них. Исходный код этих расширений можно использовать, чтобы узнать больше о программировании с PyQGIS, а также для того, чтобы удостовериться, что разработка не дублируется. Готовы к созданию расширения, но отсутствует идея? На странице Python Plugin Ideas собрано много идей и пожеланий!

## 14.2 Создание необходимых файлов

Ниже показано содержимое каталога нашего демонстрационного расширения:

```
PYTHON_PLUGINS_PATH/
  testplug/
    __init__.py
    plugin.py
    metadata.txt
    resources.qrc
    resources.py
    form.ui
    form.py
```

Для чего используются файлы:

- __init__.py = Точка входа расширения. Содержит общую информацию, версию расширения, его название и основной класс.
- plugin.py = Основной код расширения. Содержит информацию обо всех действиях, доступных в расширении, а также основной код.
- resources.qrc = XML-документ, созданный QT-Designer. Здесь хранятся относительные пути к ресурсам форм.
- resources.py = Понятная Python версия вышеописанного файла.
- form.ui = Интерфейс пользователя (GUI), созданный в QT-Designer.
- form.py = Конвертированная в код Python версия вышеописанного файла.
- metadata.txt = требуется в QGIS >= 1.8.0. Содержит общую информацию, версию расширения, его название и другие метаданые, используемые новым репозиторием расширений. Метаданным в metadata.txt отдается предпочтение перед методами из файла __init__.py. Если текстовый файл присутствует, именно он будет использоваться для получения этой информации. Начиная с QGIS 2.0 метаданные из __init__.py больше не будут использоваться и файл metadata.txt станет обязательным.

Здесь и вот здесь можно найти два способа автоматической генерации базовых файлов (скелета) типового Python расширения для QGIS. Кроме того, существует модуль Plugin Builder, который создает шаблон модуля прямо из QGIS и не требует соединения с Интернет. Это упростит работу и поможет быстрее начать разработку типового расширения.

## 14.3 Написание кода

### 14.3.1 __init__.py

Прежде всего, Менеджер модулей должен получить основные сведения о расширении, такие как его название, описание и т.д. Файл __init__.py именно то место, где должна быть эта информация:

```
def name():
  return "My testing plugin"

def description():
  return "This plugin has no real use."

def version():
```

```
  return "Version 0.1"

def qgisMinimumVersion():
  return "1.0"

def authorName():
  return "Developer"

def classFactory(iface):
  # загружаем класс TestPlugin из файла testplugin.py
  from testplugin import TestPlugin
  return TestPlugin(iface)
```

В QGIS 1.9.90 модули могут быть помещены не только в меню Модули, но и в меню Растр, Вектор, База данных и Web. Поэтому было введено новое поле метаданных “category”. Это поле используется в качестве подсказки для пользователей и сообщает где (в каком меню) искать модуль. Допустимыми значениями для параметра “category” являются Vector, Raster, Database, Web и Layers. Например, если модуль должен быть доступен из меню Растр, добавьте в __init__.py следующие строки:

```
def category():
  return "Raster"
```

### 14.3.2 metadata.txt

Для QGIS >= 1.8 необходимо создать файл metadata.txt (см. также) Пример :file:‘metadata.txt’:

```
; the next section is mandatory
[general]
name=HelloWorld
qgisMinimumVersion=1.8
description=This is a plugin for greeting the
    (going multiline) world
category=Raster
version=version 1.2
; end of mandatory metadata

; start of optional metadata
changelog=this is a very
    very
    very
    very
    very
    very long multiline changelog

; tags are in comma separated value format, spaces are allowed
tags=wkt,raster,hello world

; these metadata can be empty
; in a future version of the web application it will
; be probably possible to create a project on redmine
; if they are not filled

homepage=http://www.itopen.it
tracker=http://bugs.itopen.it
repository=http://www.itopen.it/repo
icon=icon.png
```

```
; experimental flag
experimental=True

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

### 14.3.3 plugin.py

Стоит сказать несколько слов о функции classFactory(), которая вызывается когда расширение загружается в QGIS. Она получает ссылку на экземпляр класса QgisInterface и должна вернуть экземпляр класса вашего расширения — в нашем случае этот класс называется‘‘TestPlugin‘‘. Ниже показано он должен выглядеть (например, testplugin.py):

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

# загружаем ресурсы Qt из файла resouces.py
import resources

class TestPlugin:

  def __init__(self, iface):
    # сохраняем ссылку на интерфейс QGIS
    self.iface = iface

  def initGui(self):
    # создаем действия для запуска расширения или его настройки
    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)

    # добавляем кнопку на панель и пункт в меню
    self.iface.addToolBarIcon(self.action)
    self.iface.addPluginToMenu("&Test plugins", self.action)

    # подключаемся к сигналу renderComplete, который посылается по завершению отрисовки карты
    QObject.connect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), self.renderTest)

  def unload(self):
    # удаляем пункт меню и кнопку на панели
    self.iface.removePluginMenu("&Test plugins",self.action)
    self.iface.removeToolBarIcon(self.action)

    # отключаемся от сигнала карты
    QObject.disconnect(self.iface.MapCanvas(), SIGNAL("renderComplete(QPainter *)"), self.renderTest)

  def run(self):
    # создаем и показываем диалог настройки или выполняем что-то еще
    print "TestPlugin: run called!"

  def renderTest(self, painter):
    # рисуем на карте, используя painter
    print "TestPlugin: renderTest called!"
```

Если используется QGIS 1.9.90 или старше и необходимо разместить модуль в одном из новых меню

(Растр, Вектор, База данных или Web), нужно модифицировать код функций initGui() и unload(). Так как эти новые пункты меню доступны только в QGIS 1.9.90, прежде всего необходимо проверить, что используемая версия QGIS имеет все необходимые функции. Если новые пункты меню доступны, мы можем разместить модуль в нужном месте, в противном случае будем использовать меню Модули как и раньше. Вот пример для меню Растр:

```
def initGui(self):
  # создаем действия для запуска расширения или его настройки
  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)

  # проверяем, доступно ли меню Растр
  if hasattr(self.iface, "addPluginToRasterMenu"):
    # меню Растр и одноименная панель доступны
    self.iface.addRasterToolBarIcon(self.action)
    self.iface.addPluginToRasterMenu("&Test plugins", self.action)
  else:
    # меню Растр отсутствует, размещаем модуль в меню Модули, как и раньше
    self.iface.addToolBarIcon(self.action)
    self.iface.addPluginToMenu("&Test plugins", self.action)

  # одключаемся к сигналу renderComplete, который посылается по завершению отрисовки карты
  QObject.connect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), self.renderTest)

def unload(self):
  # проверям доступно ли меню Растр и удаляем наши кнопки из соответствующего
  # меню и панели
  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)

  # отключаемся от сигнала карты
  QObject.disconnect(self.iface.MapCanvas(), SIGNAL("renderComplete(QPainter *)"), self.renderTest)
```

Полный список методов, которые можно использовать для размещения модуля в новых меню и на новых панелях инструментов доступен в описании API.

В расширении обязательно должны присутствовать функции initGui() и unload(). Эти функции вызываются когда расширение загружается и выгружается.

### 14.3.4 Файл ресурсов

Как видно в примере выше, в initGui() мы использовали иконку из файла ресурсов (в нашем случае он называется resources.qrc):

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

Хорошим тоном считается использование префикса, это позволит избежать конфликтов с другими расширениями или с частями QGIS. Если префикс не задан, можно получить не те ресурсы, которые нужны. Теперь сгенерируем файл ресурсов на Python. Это делается командой pyrcc4:

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

Вот и все... ничего сложного :) Если всё сделано правильно, то расширение должно отобразиться в Менеджере модулей и загружаться в QGIS без ошибок. После его загрузки на панели появится кнопка, а в меню — новый пункт, нажатие на которые приведет к выводу сообщения на терминал.

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.

## 14.4 Документация

Этот способ создания документации требует наличия Qgis версии 1.5.

Документацию к расширению можно готовить в виде файлов HTML. Модуль qgis.utils предоставляет функцию showPluginHelp(), которая откроет файл справки в браузере, точно так же как другие файлы справки QGIS.

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.

## 14.5 Фрагменты кода

Здесь собраны фрагменты кода, полезные при разработке расширений.

### 14.5.1 Как вызвать метод по нажатию комбинации клавиш

Добавьте в initGui():

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

И в unload():

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

Метод, вызываемый по нажатию F7:

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

### 14.5.2 Как управлять видимостью слоя (временное решение)

Примечание: в QGIS 1.5 появился класс QgsLegendInterface, позволяющий управлять списком слоёв легенды.

Так как в настоящее время методы прямого доступа к слоям легенды отсутствуют, в качестве временно решения для управления видимостью слоёв можно использовать решение на основе изменения прозрачности слоя:

```
def toggleLayer(self, lyrNr):
  lyr = self.iface.mapCanvas().layer(lyrNr)
  if lyr:
    cTran = lyr.getTransparency()
    lyr.setTransparency(0 if cTran > 100 else 255)
    self.iface.mapCanvas().refresh()
```

Метод принимает номер слоя в качестве параметры (0 соответствует самому верхнему) и вызывается так:

```
self.toggleLayer(3)
```

### 14.5.3 Как получить доступ к таблице атрибутов выделенных объектов

```
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 соответствует второй колонке
    else:
      layer.changeAttributeValue(int(ob[0]),1,b) # 1 соответствует второй колонке
    layer.commitChanges()
    else:
      QMessageBox.critical(self.iface.mainWindow(),"Error", "Please select at least one feature from current layer")
  else:
    QMessageBox.critical(self.iface.mainWindow(),"Error","Please select a layer")
```

Метод принимает один параметр (новое значения атрибута выделенного объекта(ов)) и вызывается как:

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

### 14.5.4 Как выполнять отладку при помощи PDB

Сначала добавьте следующий код в место, которое будет отлаживаться:

```
# для отладки используем pdb
import pdb
# устанавливаем точку останова
pyqtRemoveInputHook()
pdb.set_trace()
```

Затем запускаем QGIS из командной строки.

В Linux:

$ ./Qgis

В Mac OS X:

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

Когда приложение достигнет точки останова, консоль станет доступной и можно будет вводить команды!

## 14.6 Тестирование

## 14.7 Публикация расширения

Если после создания расширения вы решите, что оно может быть полезно и другим пользователям — не бойтесь загрузить его в репозиторий QGIS plugin repository. На этой же странице можно найти инструкции по подготовке пакета, следование которым избавит от проблем с установкой расширения через Установщик модулей. В случае, когда нужно настроить собственный репозиторий, создайте простой XML документ, описывающий все расширения и их метаданные. Пример файла можно найти на странице Python plugin repositories.

## 14.8 Примечание: настройка IDE в Windows

При использовании Linux разработка расширений не требует дополнительных настроек. Но в при использовании Windows необходимо убедиться, что и QGIS, и интерпретатор используют одни и те же переменные окружения и библиотеки. Наиболее простой и быстрый способ сделать это — модифицировать командный файл для запуска QGIS.

Если используется установщик OSGeo4W, командный файл можно найти в каталоге bin папки, куда выполнена установка OSGeo4W. Ищите что-то похожее на C:\OSGeo4W\bin\qgis-unstable.bat.

Далее будет описана настройка Pyscripter IDE. Настройка других сред разработки может несколько отличаться:

- Сделайте копию qgis-unstable.bat и переименуйте её в pyscripter.bat.
- Откройте это файл в редакторе. Удалите последнюю строку, которая отвечает за запуск QGIS.
- Добавьте строку для запуска pyscripter с параметром, указывающим на используемую версию Python. QGIS 1.5 использует Python 2.5.
- Добавьте еще один аргумент, указывающий на каталог, где pyscripter должен искать библиотеки Python, используемые qgis. Обычно это каталог bin папки, куда установлен OSGeo4W:

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

Теперь при запуске этого командного файла установятся необходимые переменные окружения и будет запущен pyscripter.

# Индексы и таблицы

- genindex
- search

# Алфавитный указатель

## Symbols

## A



## M



## P



## R