

PARTIAL SAMPLE

Gary Sherman

The PyQGIS Programmer's Guide

Extending QGIS 2.x with Python

locate
PRESS

Credits & Copyright

THE PYQGIS PROGRAMMER'S GUIDE EXTENDING QGIS 2.X WITH PYTHON

by Gary Sherman

Published by Locate Press LLC

COPYRIGHT © 2014 LOCATE PRESS LLC

ISBN: 978-0989421720

ALL RIGHTS RESERVED.

Direct permission requests to gsherman@locatepress.com or mail:
Locate Press LLC, PO Box 671897, Chugiak, AK, USA, 99567-1897

Cover Design Julie Springer

Interior Design Based on Tufte-L^AT_EXdocument class

Publisher Website <http://locatepress.com>

Book Website <http://locatepress.com/ppg>

No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

6

Using the Console

The QGIS Python console is great for doing one-off tasks or experimenting with the API. Sometimes you might want to automate a task using a small script, and do it without writing a full blown plugin.

In this chapter we'll take a look at using the console to explore the workings of PyQGIS, as well as doing some real work.

6.1 Console Features and Options

Let's take a quick tour of the console features and options. With QGIS running, open the console using the `Plugins->Python Console` menu. Figure 6.1, on the following page shows the console right after opening it. Normally when you first open the console, it is docked near the bottom of the QGIS window; in our example, we have undocked it to make the window a little bigger.

The lower panel of the console is the input area; results of input are displayed in the upper panel.

On the left of the console you'll see a toolbar that contains the following tools, top to bottom:



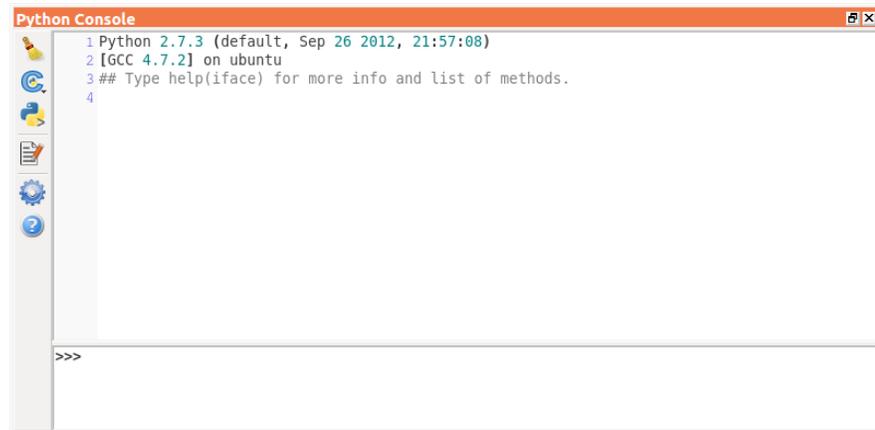
Clear console: Clears the console of all previous commands and output



Import class: Import classes from the *processing*, *PyQt4.QtCore*, or

62 CHAPTER 6. USING THE CONSOLE

Figure 6.1: The Python Console



PyQt4.QtGui module by selecting one from the popup list

 **Run command:** Run the current command in the input area

 **Show editor:** Toggle the visibility of the editor

 **Settings:** Configure the console behavior

 **Help:** Open the help window for the console

Console Options

Clicking the *Settings* button brings up the console options dialog as shown in Figure 6.2, on the next page.

You can set the font for both the Console and the Editor, as well as enabling autocompletion. For the Editor, you can enable the object inspector which allows you to get a nice view of your code.

Lastly, you can choose to use the preloaded API files or untick the box and manually add your own. Generally you can stick with the preloaded files as they contain the information needed for autocompletion in the PyQt and QGIS API.

6.2 *Using the Console Editor*

The Console Editor provides an environment in which you can edit code, providing syntax highlighting and autocompletion. This can be useful for

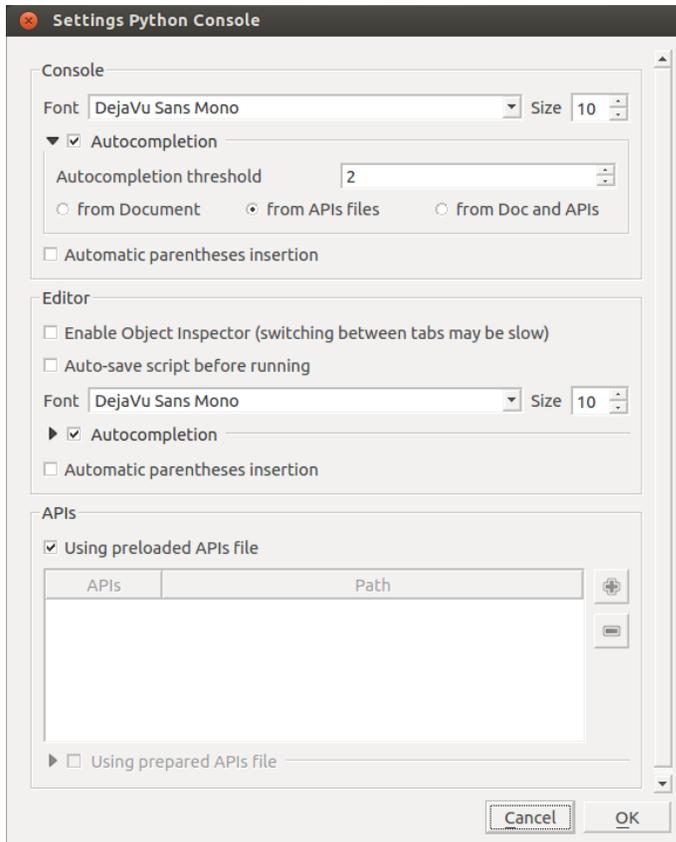


Figure 6.2: Python Console Settings

writing concise scripts, prototyping, and testing. Usually you will want to use your favorite editor or IDE when working on a larger project.

Let's look at a few of the features found in the editor.

The Toolbar

The Console Editor toolbar is arranged vertically, with the following tools top to bottom:

 **Open file:** Open a Python script located on disk. Multiple files can be opened, each assigned to a new tab.

 **Save:** Save the edited script

 **Save As...:** Save the current script to a file with a new name or location

 **Find Text:** Search the current script for occurrences of a given text string

 **Cut:** Cut the selected text to the clipboard

 **Copy:** Copy the selected text to the clipboard

 **Paste:** Paste the contents of the clipboard at the current cursor location

Comment: Comment out the current line or a selected set of lines

Uncomment: Uncomment the current line or a selected set of lines

 **Object Inspector:** Open the object inspector to show a hierarchy of classes, methods, and functions in the current script

 **Run script:** Run the current script in the console

Loading, Editing, and Running a Script

The `simple_point.py` script can be found in http://locatepress.com/files/pyqgis_code.zip, which includes all the code samples in the book.

Let's load the `simple_point.py` script that we looked at in Chapter 2, Python Basics, on page 17 and take a look at using it in the Console Editor.

Figure 6.3, on the next page shows our script loaded into the editor and the Object Inspector panel visible. Here's what we did to get to the point shown in the figure:

1. Open the Python Console
2. Click on the *Show Editor* button
3. Click *Open file* and load `simple_point.py`
4. Using the Editor, add the *my_function* function to the bottom of `simple_point.py`
5. Click *Save* to save the file
6. Open the Object Inspector and click the '+' to expand the *Point* node
7. Click the *Run script* button
8. Enter some commands in the console

```
>>> my_function()
```

```
'this does nothing'
>>> p = Point()
>>> p.draw()
drawing the point
>>> p.move(100, 100)
moving the point
```

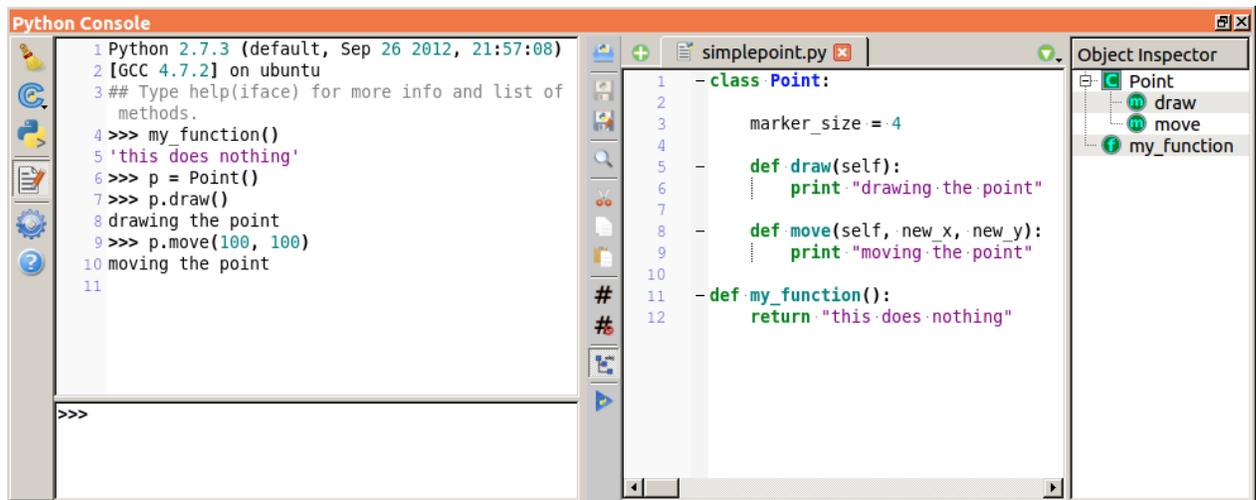


Figure 6.3: A Simple Script Loaded in the Editor

Normally I wouldn't mix class definitions and non-class functions in the same source file, but we did it here to illustrate the features of the Object Inspector. You can see that our class is listed, as well as the class methods and our new function. This is useful in navigating your code—the real benefit being when your code exceeds a few dozen lines.

When we click *Run script* in the Editor, it executes our code in the console. Since our code just defines a class and one function, there is no output in the console, however our class and function are now available for use.

In the console panel (left side), you can see the output from our commands. We'll take another look at this when we get to Section 7.1, *Standalone Scripts in the Console*, on page 73.

Now that we have an overview of the console and editor, let's put it to use. In the Introduction, we used the console to manipulate the map view in QGIS using methods exposed by the `qgis.util.iface` object. We'll take it a bit further now to actually load some data and work with the interface.

6.3 Loading a Vector Layer

To begin, let's load a shapefile into QGIS using the console. To do this, we will use the `world_borders.shp` shapefile from the sample dataset.

With QGIS running, open the console using the `Plugins->Python Console` menu. Since the `qgis.core` and `qgis.gui` modules are imported at startup, we can start using the API immediately.

To load the shapefile we will use the `QgsVectorLayer` class by creating an instance of it and passing the path to the shapefile and the data provider name. If you recall, we took a look at the documentation for `QgsVectorLayer` in Section 5.1, Finding the Documentation, on page 49:

```
QgsVectorLayer (QString path=QString::null,
                QString baseName=QString::null,
                QString providerLib=QString::null,
                bool loadDefaultStyleFlag=true)
```

To refresh your memory, the parameters are:

path:

The full path to the layer

basename:

A name to be used in the legend

providerLib:

The data provider to be used with this layer

loadDefaultStyleFlag:

Use the default style when rendering the layer. A style file has a `.qml` extension with same name as the layer and is stored in the same location.

First we create the layer in the console:

```
wb = QgsVectorLayer('/data/world_borders.shp', 'world_borders', 'ogr')
```

It is possible to create a vector layer that isn't valid. For example, we can specify a bogus path to a shapefile:

```
>>> bogus = QgsVectorLayer('/does/not/exist.shp', 'bogus_layer', 'ogr')
>>> bogus
<qgis.core.QgsVectorLayer object at 0x1142059e0>
```

Notice there is no complaint from QGIS, even though the shapefile doesn't exist. For this reason, you should always check the validity of a layer before adding it to the map canvas:

```
>>> bogus.isValid()  
False
```

If the *isValid()* method returns False, there is something amiss with the layer and it can't be added to the map.

Getting back to our valid, `world_borders` layer, you'll notice nothing happened on the map canvas. We created a layer, however, for it to draw, we need to add to the list of map layers. To do this, we call a method in the *QgsMapLayerRegistry*:

```
QgsMapLayerRegistry.instance().addMapLayer(wb)
```

Once we do that, the layer is drawn on the map as shown in Figure 6.4.

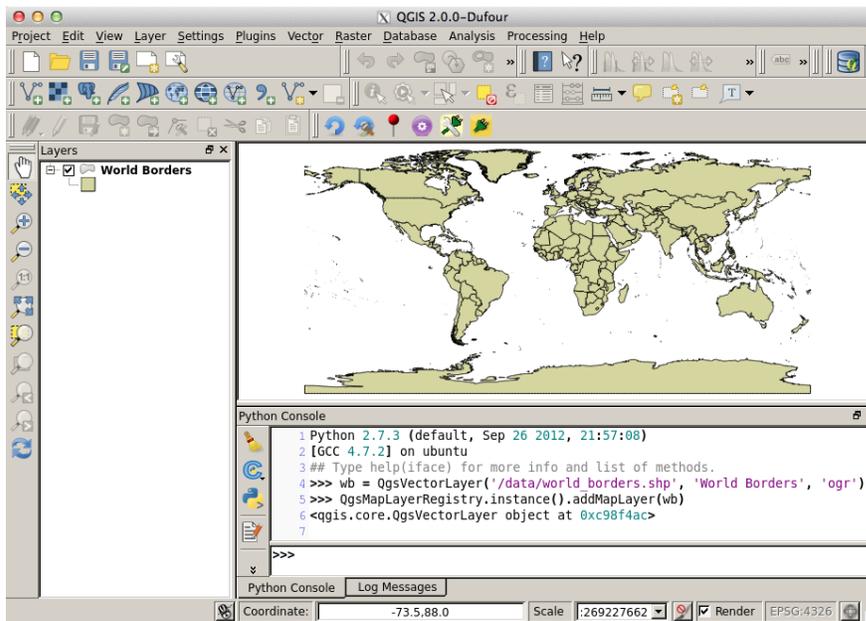


Figure 6.4: Using the Console to Load a Layer

Putting it all together, we have:

```
wb = QgsVectorLayer('/data/world_borders.shp', 'world_borders', 'ogr')  
if wb.isValid():  
    QgsMapLayerRegistry.instance().addMapLayer(wb)
```

If we want to remove the layer, we use the `removeMapLayer()` method and the layer id as an argument:

```
QgsMapLayerRegistry.instance().removeMapLayer(wb.id())
```

The layer is removed from both the map canvas and the legend, then the map is redrawn.

6.4 Exploring Vector Symbology

When you load a vector layer, it is rendered using a simple symbol and a random color. We can change the way a loaded layer looks by modifying the attributes of the symbol.

First let's load our `world_borders` layer:

```
>>> wb = QgsVectorLayer('/data/world_borders.shp', 'world_borders', 'ogr')
```

Next we get a reference to the renderer:¹⁵

```
>>> renderer = wb.rendererV2()
>>> renderer
<qgis.core.QgsSingleSymbolRendererV2 object at 0x114205830>
```

¹⁵ Prior to version 2.0, QGIS had both a “new” and “old” rendering engine. In the API, the new is referred to as `V2`.

Our layer is rendered using a `QgsSingleSymbolRendererV2` which in our case, is a simple polygon fill.

To get the symbol, we use:

```
>>> symbol = renderer.symbol()
```

To get a bit of information about the symbol, we can use the `dump()` method:

```
>>> symbol.dump()
u'FILL SYMBOL (1 layers) color 134,103,53,255'
```

The output shows us that our layer is rendered using a fill symbol (which we already knew) using a color with RGB values of 134, 103, 53 and no transparency. Let's change the color to a dark red and refresh the map:

```
>>> from PyQt4.QtGui import QColor
>>> symbol.setColor(QColor('#800000'))
```

Let's analyze what we did here. To be able to change the color, we imported the `QColor` class from the `PyQt4.QtGui` module. The `setColor` method takes

a *QColor* object as an argument.

`QColor('#800000')` creates a *QColor* object using a hex triplet string. When we pass this to *setColor* the fill color for our layer is set to a dark red. In Qt, there are many ways to create a *QColor*, including using twenty predefined colors that are accessible by name. Here are some ways you can create a valid *QColor*:

- `QColor(Qt.red)`
- `QColor('red')`
- `QColor('#ff0000')`
- `QColor(255,0,0,255)`

If you try the first method, you'll get an error because we haven't imported Qt. The fix is to import it prior to referencing *Qt.red*:

```
from PyQt4.QtCore import Qt
```

The last method is interesting in that it includes a value for the alpha-channel (transparency). Using this method of creating the color, we can also set the transparency of the layer. Each of these methods is described in the *QColor* documentation.

You'll notice that nothing happens on the map canvas when we change the color. We must tell QGIS to update the map canvas to reflect the changes to our layer:

```
>>> iface.mapCanvas().refresh()
```

This should redraw the map and our layer is now filled with a dark red color.

If nothing happened, it is likely you have render caching set to speed up map redraws. There are a couple of ways to deal with this—one is to always invalidate the cache prior to doing the refresh:

```
>>> wb.setCacheImage(None)
>>> iface.mapCanvas().refresh()
```

The other is to test to see if there is a cached image for the layer and, if so, invalidate it and then refresh:

```
>>> if wb.cacheImage() != None:
...     wb.setCacheImage(None)
...     iface.mapCanvas().refresh()
```

In practice, the first method should work in all situations, even if it is a bit “tacky” if render caching is not enabled.

Now that the layer is rendered in our new color, take a look at the legend—it still shows the previous color. To update it we need to call the *refreshLayerSymbology* method of the legend interface:

```
iface.legendInterface().refreshLayerSymbology(wb)
```

6.5 Loading a Raster Layer

Loading rasters is similar to loading a vector layer, except we use the *QgsRasterLayer* class:

```
QgsRasterLayer (const QString &path,
                const QString &baseName=QString::null,
                bool loadDefaultStyleFlag=true)
```

The parameters are the same as *QgsVectorLayer*, except we don’t need to provide a data provider name---all rasters use GDAL.

In this example, we’ll load `natural_earth.tif` raster into QGIS, display it, and then remove it using the console.

See `natural_earth.txt` in the sample data set for instructions on obtaining the Natural Earth raster.

To create the raster layer and add to the map, enter the following in the Python console:

```
>>> nat_earth = QgsRasterLayer('/data/natural_earth.tif',
... 'Natural Earth')
>>> QgsMapLayerRegistry.instance().addMapLayer(nat_earth)
<qgis.core.QgsRasterLayer object at 0x11483d8c0>
```

This creates the raster layer and adds it to *QgsMapLayerRegistry*. Note that the method used to add both vector and raster layers is the same: *addMapLayer*. We don’t have to tell the registry what type of layer we are loading since *QgsVectorLayer* and *QgsRasterLayer* are both a “type” of *QgsMapLayer*.¹⁶

¹⁶ In object oriented speak, they are child classes of *QgsMapLayer*.

Removing a raster layer is done the same way you remove a vector layer:

```
>>> QgsMapLayerRegistry.instance().removeMapLayer(nat_earth.id())
```

To get the full picture, see <http://locatepress.com> to purchase the PDF or print copy.