Workshop Overview

Automated QGIS project creation and manipulation using PyQGIS

What are we going to cover in this workshop

- Basics of Python and QGIS
- Documentation and reference guides
- Creating, saving and restoring projects
- Setting project metadata and other properties
- Loading layers, adding them to projects
- Setting project layer structures
- Applying styles to layers

The Python Console

Exploring the Console

The Python Console and Python script editor is a great way to explore the QGIS api in order to try out the logic of your code before adding it to your scripts.

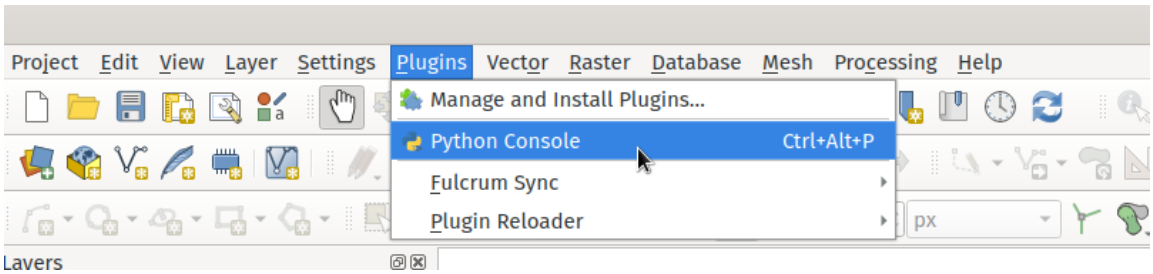
We won't spend long in here, just enough to try out some basic QGIS APIs and get used to the UI.

Exercise: Using the Console

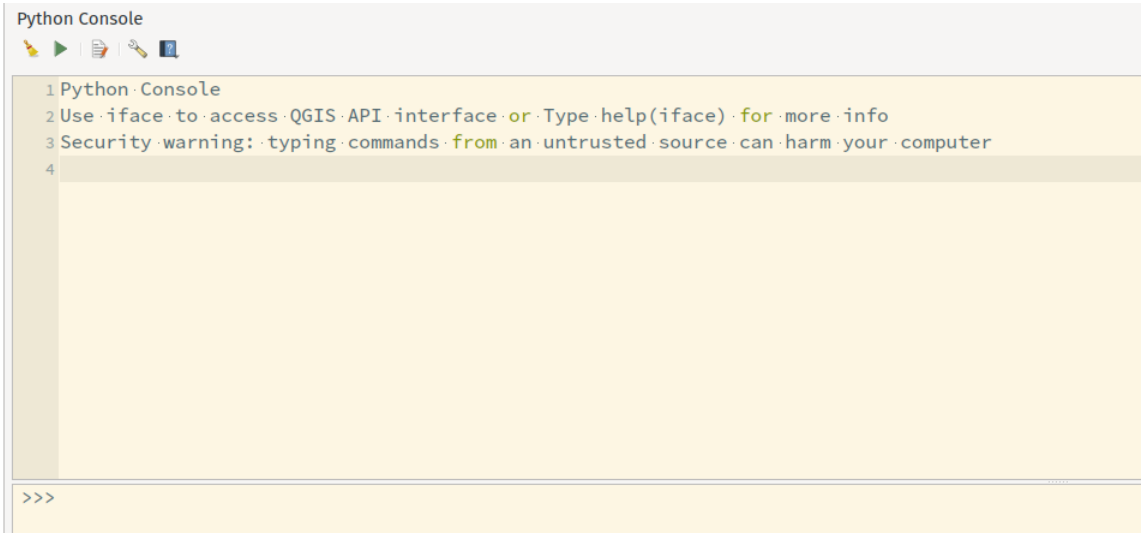
Open the console using `Plugins -> Python Console`

Running Commands

Inside the console we can run any commands we want to try. This is a full interactive Python console so any Python code will work here. To open the Python console, select "Python Console" from the Plugins menu:



The Python Console opens as a new dock at the bottom of the QGIS window. Commands can be entered in the bottom part of this dock, and output from Python will be shown at the top of the dock:



Exercise: First Python Commands

Try something basic like the following and hit enter

```
print("Python rocks")
```

The output is:

```
Python rocks
```

Exercise: First PyQGIS Commands

Let's try some QGIS commands. Try each one and check the results

```
print(Qgis.QGIS_RELEASE_NAME)
```

```
print(iface.activeLayer().name())
```

```
print(iface.activeLayer().featureCount())
```

Intro to the `iface` Object

QGIS exposes a object called `iface` (QgisInterface). This object exposes methods that you can use to make some more complex operations easier, plus methods for interacting with the current QGIS application window. It's a wrapper around some of the lower level PyQGIS functionality. One example is `iface.newProject()`.

Exercise: New Project from PyQGIS

Run the following command to create a new project

```
iface.newProject()
```

Exercise: Access the Main Canvas

We can access the main canvas and its scale function from here.

```
iface.mapCanvas().scale()
iface.mapCanvas().zoomScale(1000000)
iface.mapCanvas().extent()
iface.mapCanvas().center()

iface.mapCanvas().setCenter(QgsPointXY(..., ...))
iface.mapCanvas().refresh()
```

A Warning

While it's very useful for some scripts, it's important to note that the iface object is **only** available to scripts which are run in the main QGIS window. You cannot use iface in standalone scripts which are run from outside of QGIS! (This limitation also applies to scripts which are run using the qgis_process standalone tool).

Instead, we need to use the lower level PyQGIS API directly...

Simple Tasks

Loading a Vector Layer

Every layer in QGIS has some mandatory properties which dictate where to obtain the layer's data and how QGIS should read this data.

These properties include: - A "source". This is an encoded string which includes all the necessary details to locate the information the layer points to. You can see this by opening a layer in the QGIS application, and then double-clicking it in the layer tree. Look under the "Information" tab. - A layer's "data provider". This is basically the "driver" which QGIS will use to open the layer, e.g. "GDAL", "WMS", "Postgres", etc. - The layer type, e.g. "raster", "vector", "point cloud", "mesh", etc. Depending on the layer type we use different PyQGIS classes to create layers. For example QgsRasterLayer, QgsVectorLayer.

To construct a layer in QGIS, we must pass the source and driver name to the layer class constructor. E.g. to load a vector layer from a ShapeFile, using the OGR driver:

```
source = 'c:/Training/data/test_layer.shp'
provider = 'ogr'
my_layer = QgsVectorLayer(source, 'My New Layer', provider)
```

Warning: data provider names are case-sensitive. In this case it's `ogr`, not `OGR`. You can see a full list of the correct provider names by typing `QgsProviderRegistry.instance().providerList()` into the Python console.

The layer will load, but you won't see it in your project.

It's always good practice to check that layers have loaded correctly, by calling `isValid()` on them:

```
assert my_layer.isValid(), 'The layer could not be loaded correctly'
```

(`assert` means throw an error if it's not true!)

Querying Layers

Some simple tasks we can do on a vector layer include:

```
my_layer.featureCount()
my_layer.name()
my_layer.setName(...)
my_layer.extent()
my_layer.crs()
```

Loading a Raster Layer

Raster layers have source strings, just like vector layers! In this case we'll load a TIFF file using the gdal provider:

```
source = 'c:/Training/data/aerial.tif'
provider = 'gdal'
raster_layer = QgsRasterLayer(source, "My Raster Layer", provider)
assert raster_layer.isValid()
```

Some useful methods for querying raster layers:

```
raster_layer.width()
raster_layer.height()
raster_layer.bandCount()
raster_layer.bandName(...)
raster_layer.rasterUnitsPerPixelX()
raster_layer.rasterUnitsPerPixelY()
raster_layer.dataProvider().sample(point=QgsPointXY( 1,2 ), band=1)
```

Building Blocks of PyQGIS

Python in QGIS is actually built up of multiple different parts. From the bottom level up, we have:

1. Python (version 3)
2. The Python standard library
3. PyQt
4. PyQGIS

Lets dig into these.

Python

At the lowest level sits the Python language. This is responsible for all the core functionality, including many things we've already used:

- program logic (like `if` statements, `for` loops, etc)
- variable handling (`my_value = 5`, `my_new_value = my_value + 2`)

- statements like `print`, `assert`, `break`

QGIS uses the latest Python version – Python 3. Be careful when searching online for sample code and tips that you include "python 3" in your search terms, as results from earlier Python versions may not work!

The Python Standard Library

Much of Python's power comes from how easy it is to access powerful tools. These are provided via Python "libraries". There's hundreds of thousands of libraries you can use within your scripts, usually with just a couple of lines of code! They include everything from simple libraries for file operations through to massively complex libraries for machine learning.

All Python installs include the "Python Standard Library". This provides hundreds of libraries "out of the box", aimed at addressing common use cases. Some tools provided in the "Python Standard Library" include:

- The `os` module, for file system operations like renaming files
- The `datetime` module, for date and time calculations and handling
- The `math` module, for mathematical calculations

Often, it's as easy as typing `import math` to include this functionality in your scripts!

Documentation for the standard library is available at https://docs.python.org/3/library/. (Although often it's more helpful to Google and get more readable tutorials!)

PyQt

QGIS is built using the cross-platform "Qt" library. Qt provides rich, powerful functionality for building cross-platform applications and interfaces. It is exposed to Python through the "PyQt" library.

You can usually identify parts of Python code which are coming from the PyQt library because they all start with a "Q" prefix. E.g. `QLabel`, `QDateTime`, `QPushButton`.

Here's an example of a simple Python script which uses the PyQt library to make a user interface:

```
from PyQt5 import *

container = QWidget()
layout = QGridLayout()

my_label = QLabel()
my_label.setText('Nothing has happened')
layout.addWidget(my_label)

my_button = QPushButton()
my_button.setText('Click me!')
layout.addWidget(my_button)

container.setLayout(layout)
container.show()
```

Run the script in the console and you'll see a simple dialog. All the widgets you see here (and all throughout QGIS!) are provided by the Qt library. If you want to include a GUI in your PyQGIS scripts, you'll be using PyQt to achieve it.

Let's make the dialog interactive. Add this to the end of your script and re-run it:

```
def change_label():
    my_label.setText('You pressed the button!')
    my_button.hide()

my_button.clicked.connect(change_label)
```

PyQGIS documentation is available at https://www.riverbankcomputing.com/, but it's not very helpful. Better to view the underlying Qt library documentation (which is fantastic): https://doc.qt.io/qt-5/classes.html. It's for c++, but still explains a lot. E.g. https://doc.qt.io/qt-5/qlabel.html#text-prop explains exactly what `my_label.setText(...)` does.

PyQGIS

Lastly, we have the PyQGIS library. PyQGIS exposes all the spatial functionality which is available through QGIS to Python. You can usually identify parts of scripts which are coming from PyQGIS by their "Qgs" prefixes, (e.g. `QgsFeature`, `QgsVectorLayer`, `QgsProject`).

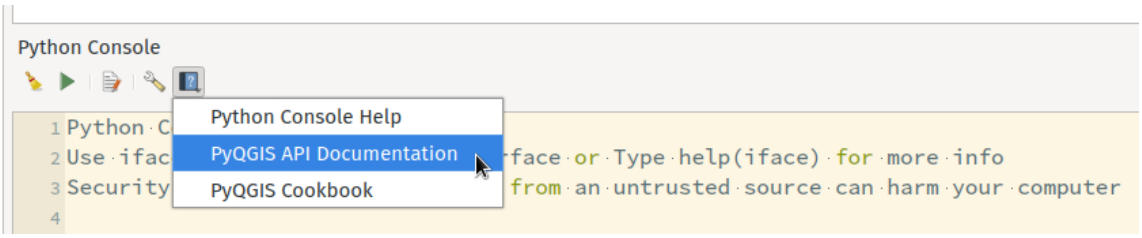
PyQGIS exposes both the "core" functionality for handling spatial data along with the "gui" functionality used by QGIS itself.

The `qgis.core` module includes all the lower-level parts of a GIS: layer handling, features, geometries, etc. The `qgis.gui` library provides re-usable widgets you can use in your scripts - like map canvases, widgets for selecting map layers, or editing print layouts.

(There's also the `qgis._3d`, `qgis.analysis`, `qgis.processing` and `qgis.server` modules, but they are less commonly used!)

Important If you're writing a standalone script for use outside of the QGIS application, you'll only have access to some of the qgis modules (e.g. `qgis.core`). The `qgis.gui` module is only available when you're creating a window-based application.

The PyQGIS API documentation is generally quite helpful, and is available at https://qgis.org/pyqgis/master/. You can also directly open the API documentation using the shortcut action in the Python Console toolbar:



Working with Projects

A "project" in QGIS is a document which defines sets of map layers and their styles, layouts, annotations, canvases, etc. In PyQGIS, the QgsProject class encapsulates everything to do with a QGIS project.

Active project vs "temporary" projects

While the main QGIS application only allows users to open a single project at once, this same limitation doesn't apply in PyQGIS and your scripts can potentially reference many projects simultaneously.

While reading the PyQGIS documentation and many examples you find online, you'll often see references to `QgsProject.instance()....`, e.g. `QgsProject.instance().addMapLayer(...)`. `QgsProject.instance()` is a special object which points to the main project shown in the QGIS window. Unlike the special `iface` object though, `QgsProject.instance()` can be used in standalone scripts. You'll generally see this referenced in various documentation and examples simply because it works everywhere and it's often easier to write a script when you can see the changes immediately applying to the project open in your QGIS window.

(Formally, `QgsProject.instance()` is a type of programming pattern called a "singleton").

Let's set some properties on the current project in a QGIS desktop window by entering some commands in the Python Console:

```
active_project = QgsProject.instance()
active_project.setTitle('My project')
active_project.title()
```

You should see the project title at the top of the window update immediately. In this case, we're modifying the `QgsProject.instance()` object, so we are modifying the user's current project.

Now, let's create another project which **isn't** the active project.

```
another_project = QgsProject()
another_project.setTitle('My other project')
another_project.title()
```

We've made a new project and stored it in the `another_project` variable, but the user won't see any visible changes -- everything is being done behind-the-scenes to the newly created "temporary" project.

Saving and restoring projects

A QGIS `.qgs` project is just an XML file (or in the case of `qgz` files, a zipped XML file). While the QgsProject class is used to manipulate a project in PyQGIS, the changes are **not** immediately written to disk or saved anywhere. We have to manually trigger a save to disk, by using the QgsProject.write method

```
another_project.write('/home/nyall/test_project.qgs')
```

We can then open this project in QGIS to confirm that our manually set project title has been remembered!

To restore a project we use QgsProject.read

```
my_project = QgsProject()
my_project.title()

my_project.read('/home/nyall/test_project.qgs')
my_project.title()
```

Note: While a QGIS project IS just XML, we never recommend directly editing or reading this XML in scripts. The XML format is not stable between versions and trying to manually parse it in scripts will never be stable. It's better to use the PyQGIS api to manipulate the project directly instead, and leave the XML handling to QGIS itself.

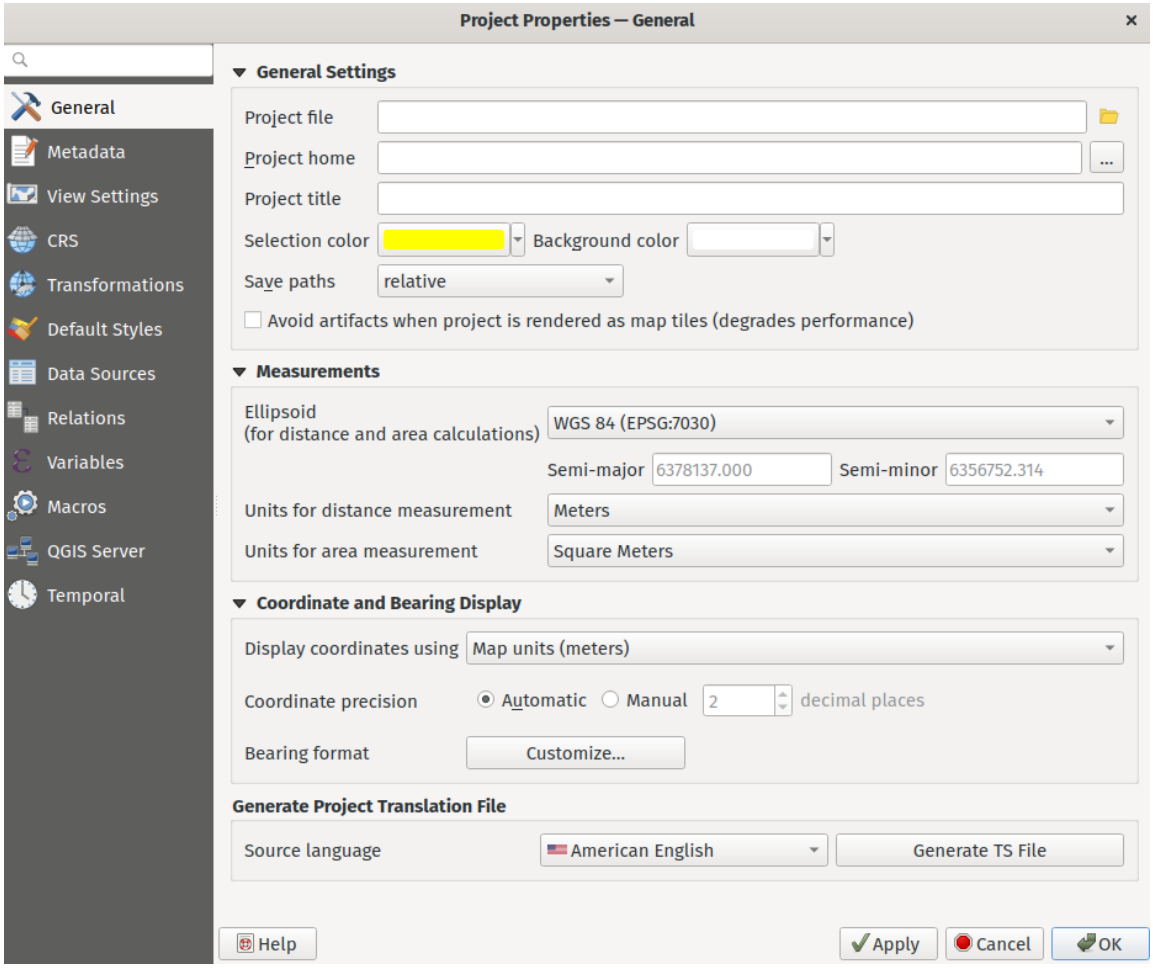
Resetting a project

A handy method to know is the QgsProject.clear method, which resets a project back to the default "new" project state:

```
my_project.clear()
my_project.title()
```

Setting Project Properties

QGIS projects have many general properties which define how that project operates within QGIS. You can see many of these in the QGIS desktop application through the Project Properties dialog:



Let's now set some common properties for our project! We'll start by setting up some handy metadata for the project (using the QgsProjectMetadata class):

```
my_project = QgsProject.instance()

# retrieve the existing metadata
metadata = my_project.metadata()

# change some metadata properties
metadata.setAuthor('Someone')
metadata.setTitle('My project')
metadata.setAbstract('A project for showcasing PyQGIS functionality.')
metadata.setHistory(['Created using the python create_project tool on
30-03-2021'])
# note that setHistory requires a list of strings!

# store the new metadata in the project
my_project.setMetadata(metadata)
```

We can check the results via the Project Properties - Metadata tab.

Project CRS

Every QGIS project has a preset coordinate reference system (CRS) assigned to it. This CRS is used in the main map canvas, and will be the default CRS for newly created objects such as maps in a print layout.

Before we can set the project's CRS, we first need to construct a QgsCoordinateReferenceSystem object.

There's two common ways we can do this: 1. If the desired CRS is a standard CRS from an authority such as the EPSG registery, we can use its assigned authority code to create the CRS directly:

```
# WGS 84
my_crs = QgsCoordinateReferenceSystem('EPSG:4326')
# Web mercator
my_crs = QgsCoordinateReferenceSystem('EPSG:3857')
# WGS84 / UTM zone 12 N
my_crs = QgsCoordinateReferenceSystem('EPSG:32612')
```

1. If it's not a standard CRS, we can construct the CRS using a WKT definition of the system:

```
my_crs =
QgsCoordinateReferenceSystem.fromWkt('PROJCRS["Custom",BASEGEOGCRS["WGS
84",DATUM["World Geodetic System 1984"...')
```

(you can also use the older Proj syntax by `QgsCoordinateReferenceSystem.fromProj`, but that's not recommended for use in 2021).

After we've constructed a CRS, we can interrogate it using any of the QgsCoordinateReferenceSystem methods:

```
my_crs.description()
my_crs.mapUnits()
my_crs.bounds()
```

To set a project's CRS, we simply call QgsProject.setCrs:

```
my_project.setCrs(my_crs)
```

Ellipsoid and units

The choice of ellipsoid and area and length units will affect how measurements are calculated in your project. We can set these using the corresponding `QgsProject` methods:

```
my_project.setEllipsoid('EPSG:7030')
my_project.setDistanceUnits(QgsUnitTypes.DistanceKilometers)
my_project.setAreaUnits(QgsUnitTypes.AreaHectares)
```

CRS Transformations

Lastly, it can sometimes be required to setup default transformation parameters between different CRSes when layers with different reference systems are loaded into the project. This is done using the QgsCoordinateTransformContext class. (But we'll skip the gory details here!)

Working with Layers

A QGIS project acts as a container for map layers. Inside the QGIS desktop application, users can load any combination of raster, vector, mesh and point cloud layers into the project. We can do the same directly via PyQGIS!

We've already seen how to construct layers using their `source` and `provider`, eg:

```
my_vector_layer = QgsVectorLayer('c:/data/training/roads.shp', 'Roads',
'ogr')
my_raster_layer = QgsRasterLayer('c:/data/training/aerial.tif', 'Aerial',
'gdal')
my_point_cloud_layer = QgsPointCloudLayer('c:/data/training/dem.laz', 'DEM',
'pdal')

# Don't forget to always check that your layers are valid after creating
them!
assert my_vector_layer.isValid()
```

Adding Layers to Projects

The simplest way to add a layer to a project is via QgsProject.addMapLayer

```
my_project = QgsProject.instance()
my_project.addMapLayer(my_layer)
```

You can also add multiple layers at once via `addMapLayers`:

```
my_project.addMapLayers([layer1, layer2, layer3])
```

Loading layers into a project like this will automatically insert them into a default position in the project's layer tree list.

Querying Layers from the Project

If you query which layers are already added to a project via `QgsProject.mapLayers`. This returns a Python dictionary of layers, with unique, autogenerated "layer IDs" as their key:

```
my_project.mapLayers()

# iterate through the Python dictionary
for layer_id, layer in my_project.mapLayers().items():
    print('layer ' + layer_id + ' is named ' + layer.name())
```

Manually Structuring Projects

If you need to control exactly how layers are added to a project (i.e. the structure of the Layers panel), you can do this through PyQGIS and the QgsProject.layerTreeRoot getter.

First, we need to disable the automatic insertion of layer into the Layer tree which we saw earlier. We do this by calling `QgsProject.addMapLayer` or `QgsProject.addMapLayers` with a `False` value as the second argument:

```
my_project = QgsProject.instance()
my_project.addMapLayer(my_layer, False)
```

The layer(s) will be added to the project, but you won't see them appear in the Layers panel yet! That's up to us to do manually now...

```
layer_tree_root = my_project.layerTreeRoot()
```

`layer_tree_root` is now a reference to the "root" (or top level) group from the Layers panel. You never actually "see" this root group in the layers panel – it's invisible to users.

Let's create a group to add our layers into:

```
my_transport_group = layer_tree_root.addGroup('transport')
# you'll see a new empty group added to the project, entitled "transport"
```

```
# add some layers to the group
my_transport_group.addLayer(layer1)
my_transport_group.addLayer(layer2)
my_transport_group.addLayer(layer3)

# add a subgroup
my_transport_roads_group = my_transport_group.addGroup('roads')
my_transport_roads_group.addLayer(layer4)
my_transport_roads_group.addLayer(layer5)
```

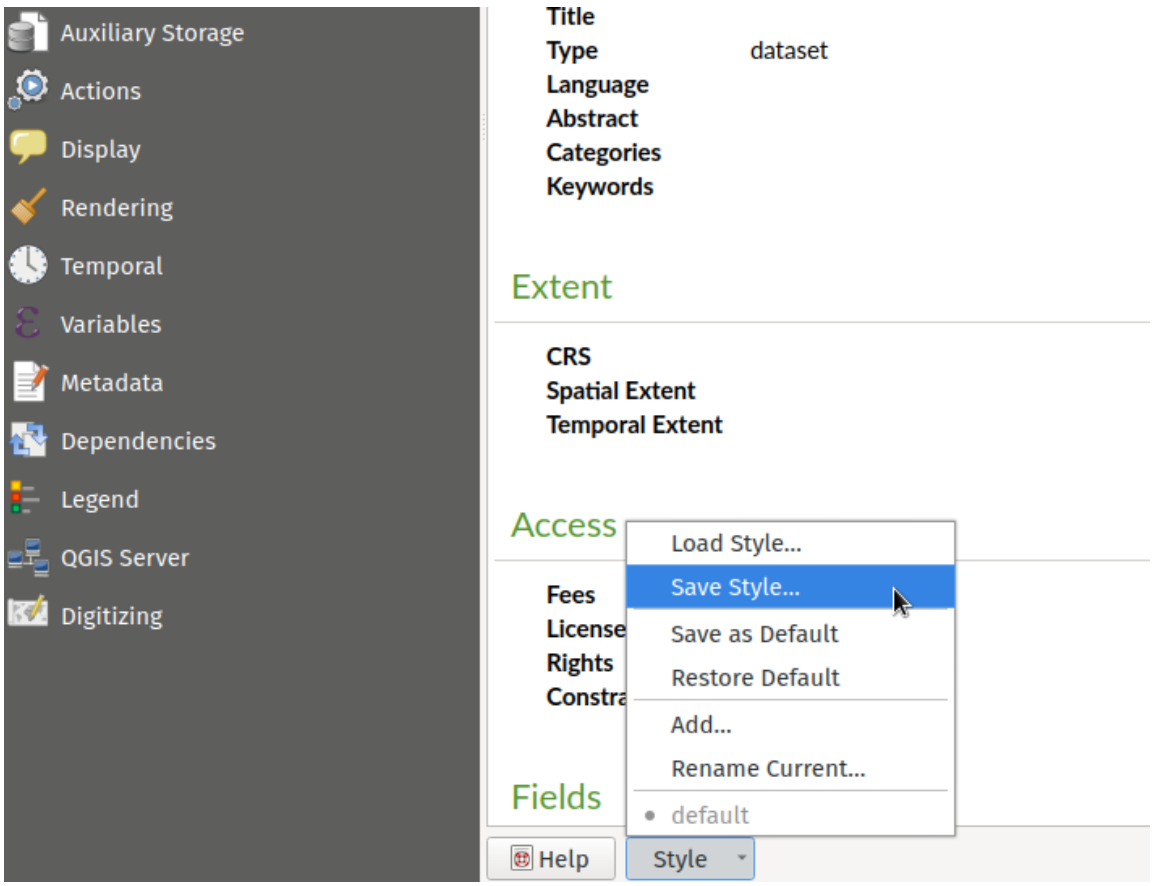
Manually adding layers to the layer tree allows us to control exactly how the project will be structured, and the order of these layers in the project.

Styling Layers

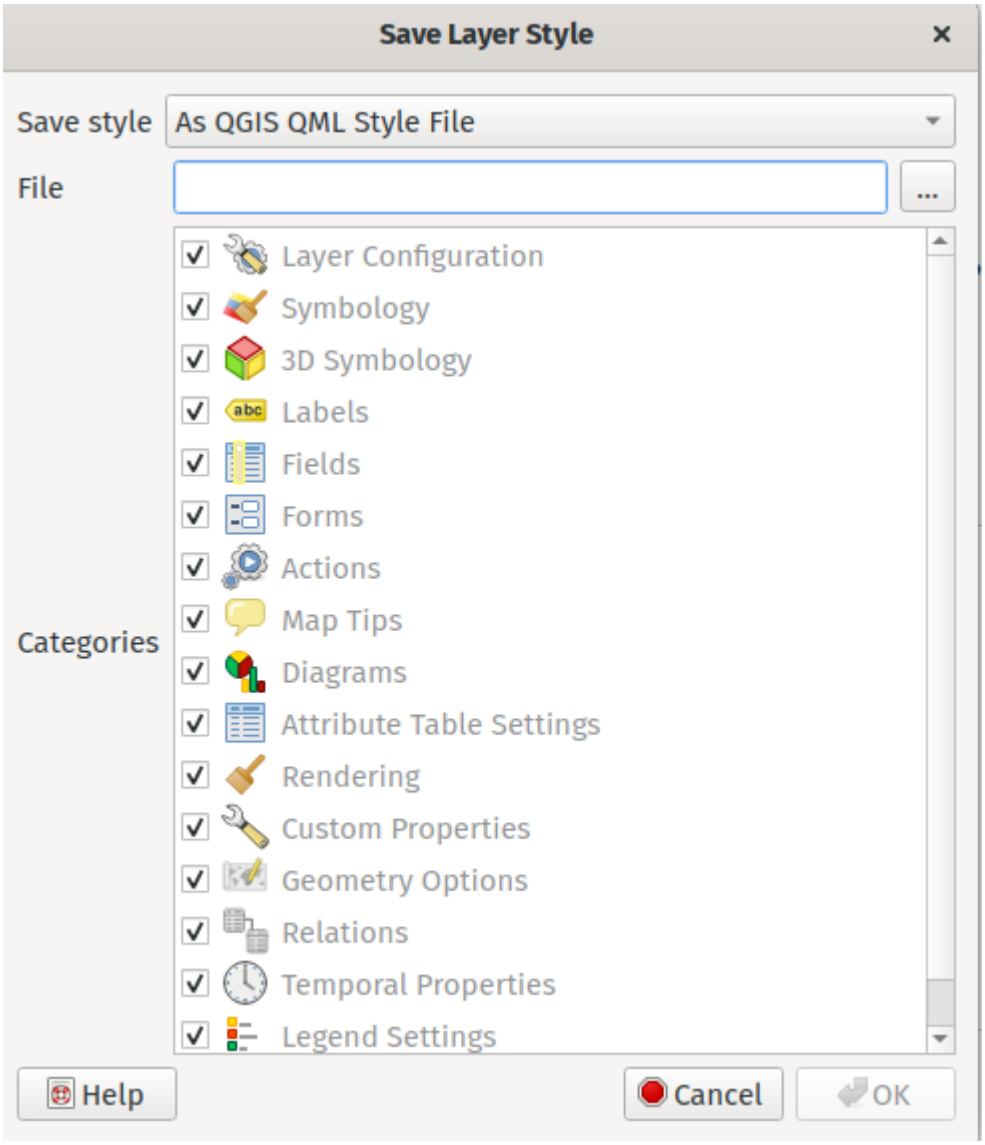
While it's possible to directly style layers using all the lower level PyQGIS classes (e.g. creating a "Categorized" renderer, populating its classes, and assigning it to a layer), instead it's preferable to short-cut this wherever possible and instead use pre-made "QML" files to assign styles to layers.

To do this, we load a layer into QGIS and use the GUI based tools to define all desired properties of the layer, including: - Layer appearance - Labeling settings - Form and widget configuration

Then, we save the layer's style out to a `QML` file type by clicking the "Style" button at the bottom of the layer properties window and selecting "Save Style".



Choose to save the style as a "QGIS QML Style File", and check whichever categories you want to save in the file:



Applying a QML file to a layer in PyQGIS

Now, in our PyQGIS script it's quite straightforward to apply the styling from a `QML` file directly to a layer. We do this using QgsMapLayer.loadNamedStyle:

```
path_to_qml = 'c:/data/rail_lines.qml'
assert my_layer.loadNamedStyle(path_to_qml)
```

The same process applies regardless of the layer type, including raster, vector, point cloud or mesh layers.

Filtering a Vector Layer

There's MANY useful methods for querying and modifying layers available through PyQGIS. You can see these by browsing through the QgsMapLayer documentation, or by looking at the layer-specific subclasses: - QgsVectorLayer - QgsRasterLayer - QgsMeshLayer - QgsPointCloudLayer - QgsAnnotationLayer

One common task is to apply a filter to a vector layer. It takes some reading through the `QgsVectorLayer` reference guide to find this, but eventually we'll be led to QgsVectorLayer.setSubsetString. Calling this method allows us to apply a SQL filter to the layer (with exact syntax varying depending on the data provider used):

```
my_layer.setSubsetString("feature_type='rail_tourist'")
```

Warning: Be VERY careful about the use of double vs single quotes. Double quotes are for field names, single quotes are for string literal values. You'll need to take care to correctly escape strings in Python when setting a layer's subset string.

We can remove the subset filter at a later stage by resetting it to an empty string:

```
my_layer.setSubsetString("")
```

Other Tasks

Creating Pyramids on a Raster Layer

An exercise:

- Looking at the QgsRasterLayer documentation, we see no mention of pyramids.
- Let's look in the QgsRasterDataProvider class instead. Every map layer in QGIS has an underlying data provider which we can access through `layer.dataProvider()`. The data provider classes expose very low level, provider specific functionality.
- Searching through QgsRasterDataProvider for pyramids gives two hits: QgsRasterDataProvider.buildPyramidsList and QgsRasterDataProvider.buildPyramids. Unfortunately the documentation for both methods is very "light"!
- It looks like we first have to build a "list" of pyramids, and then use this to call buildPyramids.

Let's try calling `buildPyramidsList` on a raster and see what we get:

```
list = my_layer.dataProvider().buildPyramidList()
print(list)
```

- Clicking on `QgsRasterPyramid` in the API docs gives a few more clues. Let's dig further:

```
for l in list:
    print(l.build, l.exists, l.level, l.xDim, l.yDim)
```

Ok, we can see that no pyramids exist, and some properties of these.

Let's try calling the second method and see what happens:

```
my_layer.dataProvider().buildPyramids(list)
```

Nothing happens, but we don't get any errors...

With a bit of trial and error we can determine that we need to set the "build" flag on the pyramids list for the levels we want!

```
for l in list:
    l.build = True
my_layer.dataProvider().buildPyramids(list)
```

This took some time to process, so things look good. The pyramids have been created!