The field of geostatistics covers a wide range of spatial statistical topics such as:

- Semi-variograms to characterize the spatial pattern in the data
- Kriging for spatial prediction
- Standard error to measure uncertainty about unsampled values

Geostatistics is a growing field of study that we use in mining, climate studies, soil science, and most environmental fields.

Why use geostatistics?

The three main tools that geostatistics provides are:

- **Semi-variograms** to model the relationship between all pairs of points.
- Kriging modeling to predict values at unsampled locations.
- Standard error to measure confidence at unsampled values.

For example, if you have soil samples at specific locations, geostatistics can answer these types of questions:

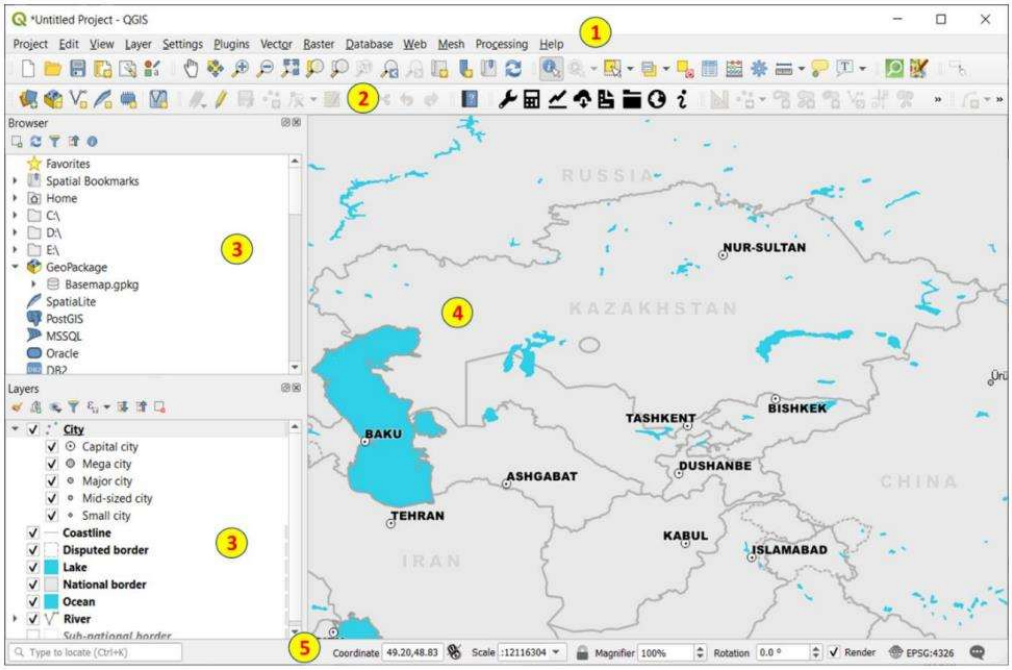
What is the forecasted amount of soil moisture at unsampled locations? How confident is the spatial prediction for the amount of soil moisture is true?

5.3 GIS programming and customization: Opening and exploring Model Builder, Python script tools, Customizing QGIS with Python

Opening and exploring Model Builder:

QGIS can be run from a User Interface (GUI) or command line and integrates tools from other free and open systems like GRASS, GDAL, SAGA and programming languages like Python and R. The GUI is has different components that can be re-arranged and personalized. Basically the main QGIS GUI types of components are:

1. Menu Bar 2. Toolbars 3. Panels 4. Map View 5. Status Bar



There are many Panels and Toolbars in QGIS that can be made visible and/or located in different parts of the GUI Windows. Just try a right click on the menu bar (2) to see the available ones. Also in the Browser or Layer Panels (3) you can make a Right click on a file to access the contextual menu that contains many options.

Downloading and exploring the data

Please first download the Testing Dataset.

Please extract the data in a folder and explore to see few Shape files (*.shp) that will be used in this demonstration. Layers are a free open dataset for the whole world produced by OpenStreetMap (OSM) https://www.openstreetmap.org.

There are many ways to access and download data from this repository, like external services (https://overpass-turbo.eu/ or https://download.geofabrik.de/) or QGIS plugins like: OSMDownloader or QuickOSM.

The layers provided contain OSM data for:

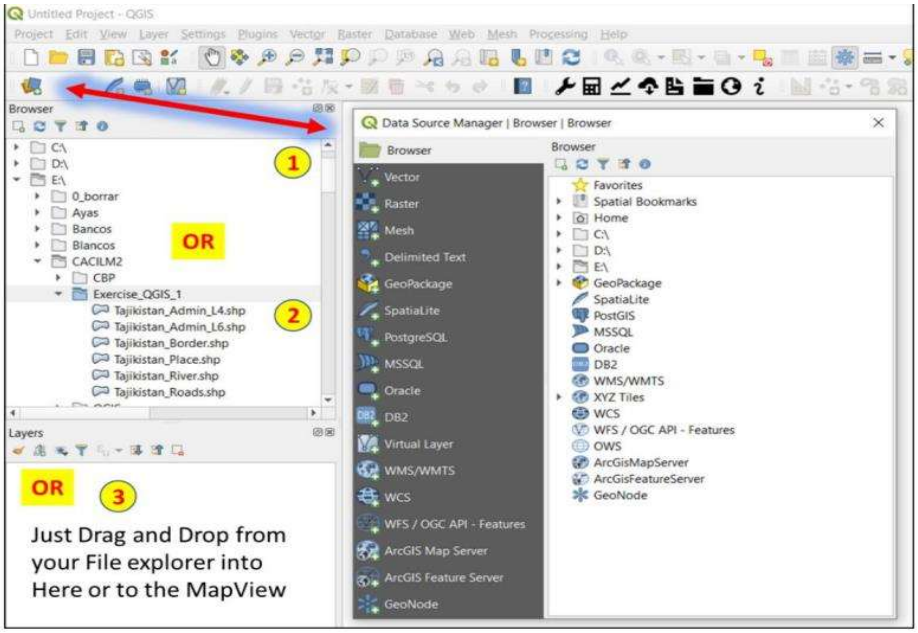
- Tajikistan Border: Country limits
- Tajikistan Admin 4: Second administrative level boundaries (province)
- Tajikistan Admin 6: Third administrative level boundaries (district)

- Tajikistan Roads: Main roads

- Tajikistan River: Main Rivers

- Tajikistan Place: Populated places

There are many different options to open a File in QGIS:

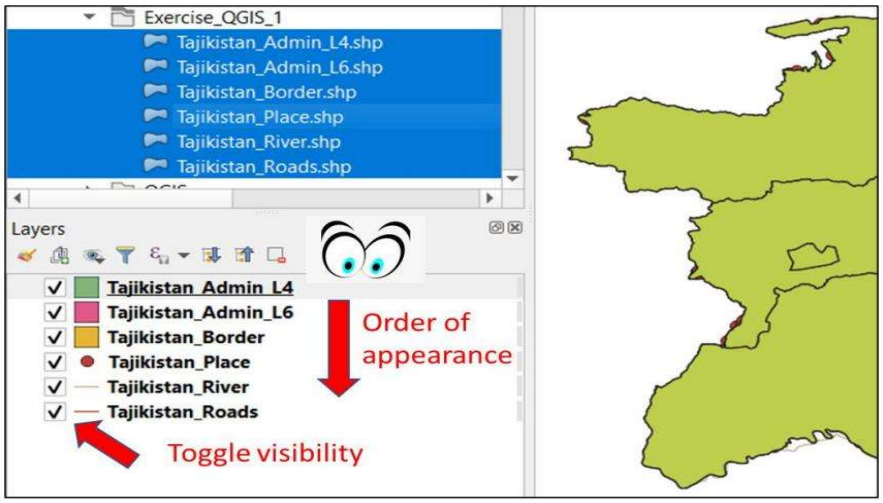


- 1.- Using the Data Source Manager (Ctrl + L) to choose the type of File and specify details.

- 2.- Using QGIS Browser and double clicking on the layer you want to open.

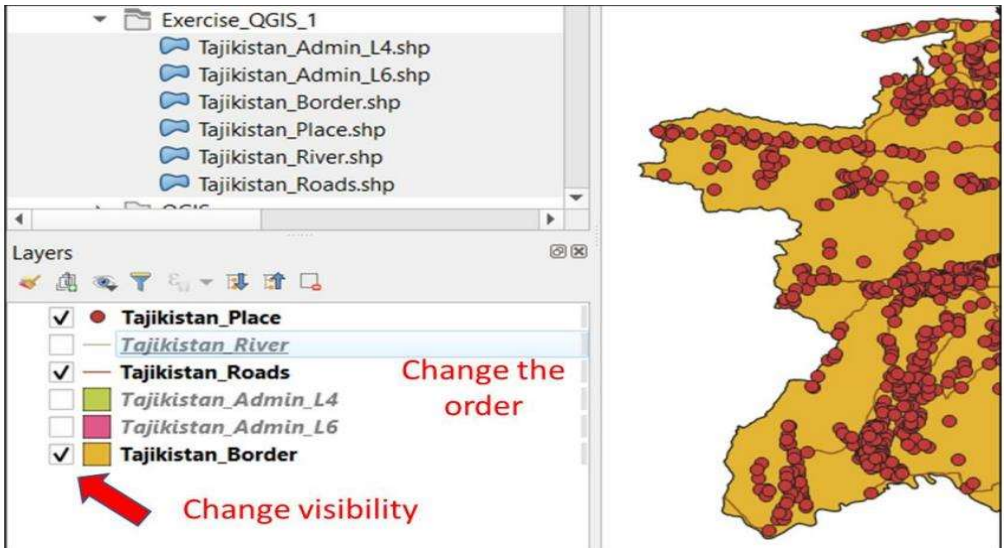
- 3.- My Favorite☺: Just Drag and Drop from your normal file explorer.

Please, open all the files provided to see their content in the Map View. By using the QGIS Browser you can select all of them and open them all at once:

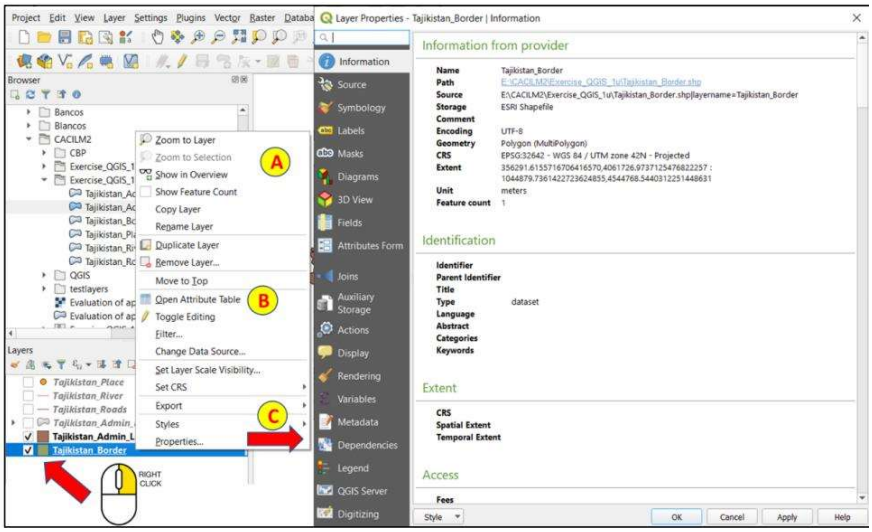


Here you have to consider a few things: • The color of the layer is assigned randomly • The order of appearance is from top to botton in the list, so if you have a polygon with solid fill on top it will cover what is below. • You can just drag and drop items in the list and toggle the visibility in order to organize your view.

Play with the order and visibility to see the layers:



After exploring the structure of the open layers to know what type of vector file we have in each case (point, line, polygon), we should proceed to explore what data we have in the databases and what their representation system is. A lot of information and functions can be obtained through the Context Menu, by Right Clicking the name of the layer:



• A.- Context Menu: has many basic functions for the specific layer. • B.- Open Attribute Table, will give us access to the database associated with the Layer • C.- Properties: Opens the layer Properties Menu: One of the most used functions for Layer management. Here you can (and we will use it a lot): - Check the layer information and Sources - Change the Symbology - Manage Labels - Etc. etc.

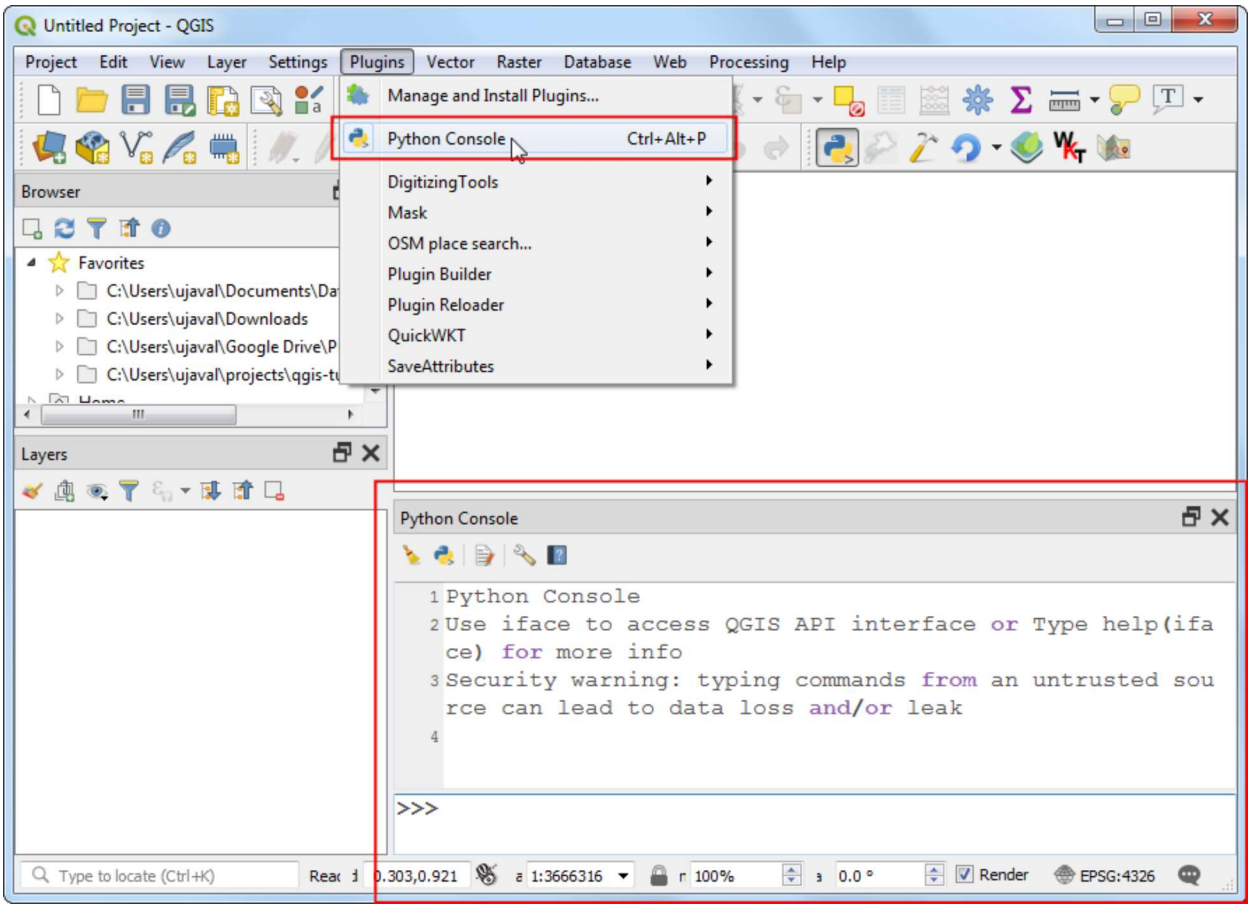
Where can you use Python in QGIS?

- Issue commands from Python Console
- Automatically run python code when QGIS starts
- Write custom expressions
- Write custom actions
- Create new processing algorithms
- Create plugins
- Create custom standalone applications

1. Hello World!

QGIS Comes with a built-in **Python Console** and a code editor where you can write and run Python code.

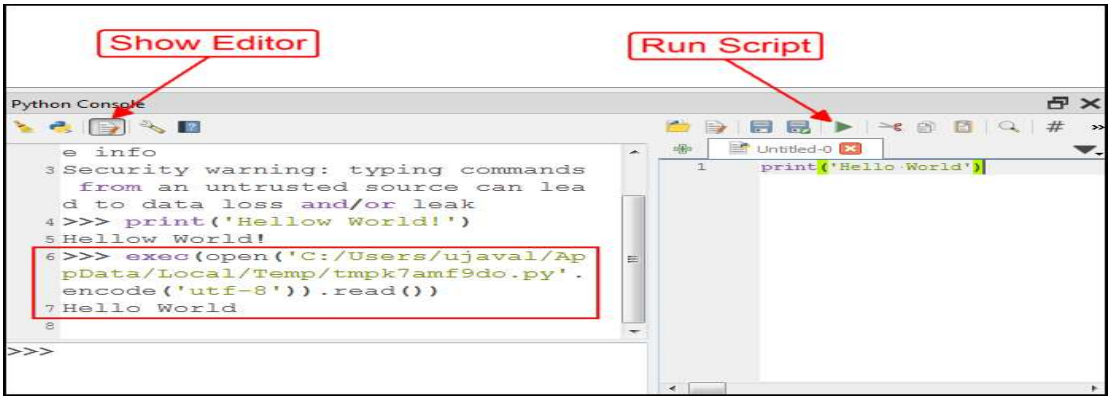
Go to **Plugins → Python Console** to open the console.



At the >>> prompt, type in the following command and press Enter.

```
print('Hello World!')
```

While console is useful for typing 1-2 lines of code or printing information contained in a variable, you should use the built-in editor for typing longer scripts or code snippets. Click the *Show Editor* button to open the editor panel. Enter the code and click the *Run Script* button to execute it. The results will appear in the console as before. If you are working on a longer script, you can also click the *Save* button in the editor to save the script for future use.

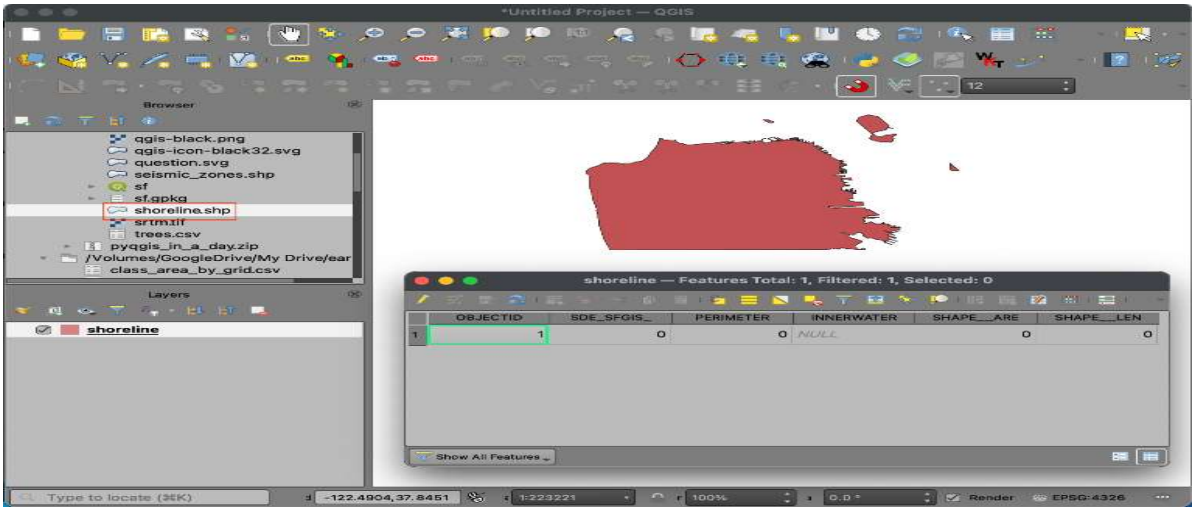


All code snippets below should be run from the Editor.

2. Hello PyQGIS!

QGIS provides a Python API (Application Programming Interface), commonly known as PyQGIS. The API is vast and very capable. Almost every operation that you can do using QGIS - can be done using the API. This allows developers to write code to build new tools, customize the interface and automate workflows. Let's try out the API to perform some GIS data management tasks.

Browse to the data directory and load the shoreline.shp layer. Open the *Attribute Table*. This layer has 6 attribute columns. Let's say we want to delete the 2nd column (SDE_SFGIS_) from the layer.



This can be done using the QGIS GUI as follows

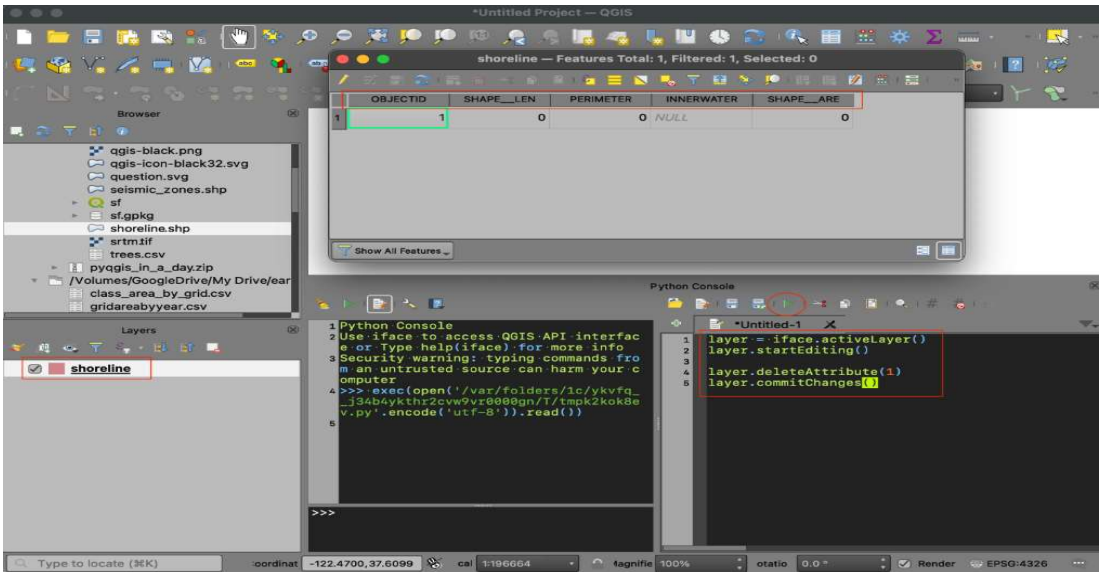
1. Right-click the shoreline layer and click *Open Attribute Table*.
2. In the Attribute Table, click the *Toggle Editing mode* button.
3. Click the *Delete field* button. Select the SDE_SFGIS_ column and click *OK*.
4. Click the *Save edits* button and click the *Toggle Editing mode* to stop editing.

QGIS Provides an API to accomplish all of this using Python code. We will now do this task - but using only Python code. Open the *Editor* and enter the following code. Click the *Run Script* button to execute it.

Make sure you have selected the shoreline layer in the *Layers* panel before running the code.

```
layer = iface.activeLayer()
layer.startEditing()
layer.deleteAttribute(1)
layer.commitChanges()
```

You will see that the 2nd column is now deleted from the attribute table.



Let's understand the code step-by-step

1. layer = iface.activeLayer(): This line uses the iface object and runs the activeLayer() method which returns the currently selected layer in QGIS. We will learn more about iface in the QGIS Interface API section. The method returns the reference to the layer which is saved in the layer variable.
2. layer.startEditing(): This is equivalent to putting the layer in the editing mode.
3. layer.deleteAttribute(1): The deleteAttribute() is a method from QgsVectorLayer class. It takes the index of the attribute to be deleted. Here we pass on index 1 for the second attribute. (index 0 is the first attribute)
4. layer.commitChanges(): This method saves the edit buffer and also disables the editing mode.

This gives you a preview of the power of the API. To harness the full power of the PyQGIS API, we must first understand how classes work.

3. Understanding Classes

Before we dive it to PyQGIS, it is important to understand certain concepts related to C++ and Python Classes. Qt as well as QGIS is written in C++ language. Functionality of each Qt/QGIS Widget is implemented as a class - having certain properties and functions. When we use PyQt or PyQGIS classes, it is executing the code in the C++ classes via the python bindings.

Here's the code to create a class called Car demonstrating the example we covered in the Classes and Objects presentation. It creates a class, initializes it to create new instances and demonstrates the concept of inheritance. A new class is defined using the word class. All classes have a function called__init__(), which is always executed when a new object is being created. There is also the keyword self which refers to the current instance of the class. The code uses the super keyword to refer to the parent class.

```
class Car:
```

```
    model = 'Civic'

    def __init__(self, color, type):
        self.color = color
        self.type = type
        self.started = False
        self.stopped = False

    def start(self):
        print('Car Started')
        self.started = True
        self.stopped = False

    def stop(self):
        print('Car Stopped')
        self.stopped = True
        sefl.started = False

# Instantiate the class
my_car = Car('blue', 'automatic')

print(my_car)

# Call a method
my_car.start()
# Check the value of an instance variable
print('Car Started?', my_car.started)

# Check the value of a class variable
print('Car model', Car.model)

# Inheritance

class Sedan(Car):

    def __init__(self, color, type, seats):
        super().__init__(color, type)
        self.seats = seats

class ElectricSedan(Sedan):

    def __init__(self, color, type, seats, range_km):
        super().__init__(color, type, seats)
        self.range_km = range_km

my_car = Sedan('blue', 'automatic', 5)
print(my_car.color)
print(my_car.seats)
my_car.start()


my_future_car = ElectricSedan('red', 'automatic', 5, 500)
print(my_future_car.color)
print(my_future_car.seats)
```

```
print(my_future_car.range_km)
my_future_car.start()
```

4. Using PyQGIS Classes

Let's start working with PyQGIS classes now. QGIS has classes for the whole range of operations - from building the user interface to doing geoprocessing. We will see how to access these classes via the PyQGIS API.

4.1 Calculating distance using PyQGIS

A basic but important operation in a GIS is the calculation of distance and areas. We will see how you can use PyQGIS APIs to compute distances.

We will compute distance between the following 2 coordinates

```
san_francisco = (37.7749, -122.4194)
new_york = (40.661, -73.944)
```

QGIS provides the class QgsDistanceArea that has methods to compute distances and areas.

To use this class, we must create an object by instantiating it. Looking at the class documentation, the class constructor doesn't take any arguments. We can use the default constructor to create an object and assign it to the d variable.

```
d = QgsDistanceArea()
```

The documentation mentions that if a valid ellipsoid has been set for the QgsDistanceArea, all calculations will be performed using ellipsoidal algorithms (e.g. using Vincenty's formulas). As we want to compute the distance between a pair of latitude/longitudes, we need to use the ellipsoidal algorithms. Let's set the ellipsoid WGS84 for our calculation. We can use the method setEllipsoid(). Remember, methods should be applied on objects, and not classes directly.

```
d.setEllipsoid('WGS84')
```

Tip: All valid ellipsoids can be found by calling QgsEllipsoidUtils.acronyms()

Now our object d is capable of performing ellipsoidal distance computations. Browsing through the available methods in the QgsDistanceArea class, we can see a measureLine() method. This method takes a list QgsPointXY objects. We can create these objects from our coordinate pairs and pass them on to the measureLine() method to get the distance. The output will be in meters. We divide it by 1000 to convert it to kilometers.

```
lat1, lon1 = san_francisco
lat2, lon2 = new_york
# Remember the order is X,Y
point1 = QgsPointXY(lon1, lat1)
point2 = QgsPointXY(lon2, lat2)
```

```
distance = d.measureLine([point1, point2])
print(distance/1000)
```

Putting it all together, below is the complete code to calculate the distance between a pair of coordinates using PyQGIS. When you run the code in the Python Console of QGIS, all the PyQGIS classes are already imported. If you are running this code from a script or a plugin, you must explicitly import the QgsDistanceArea class.

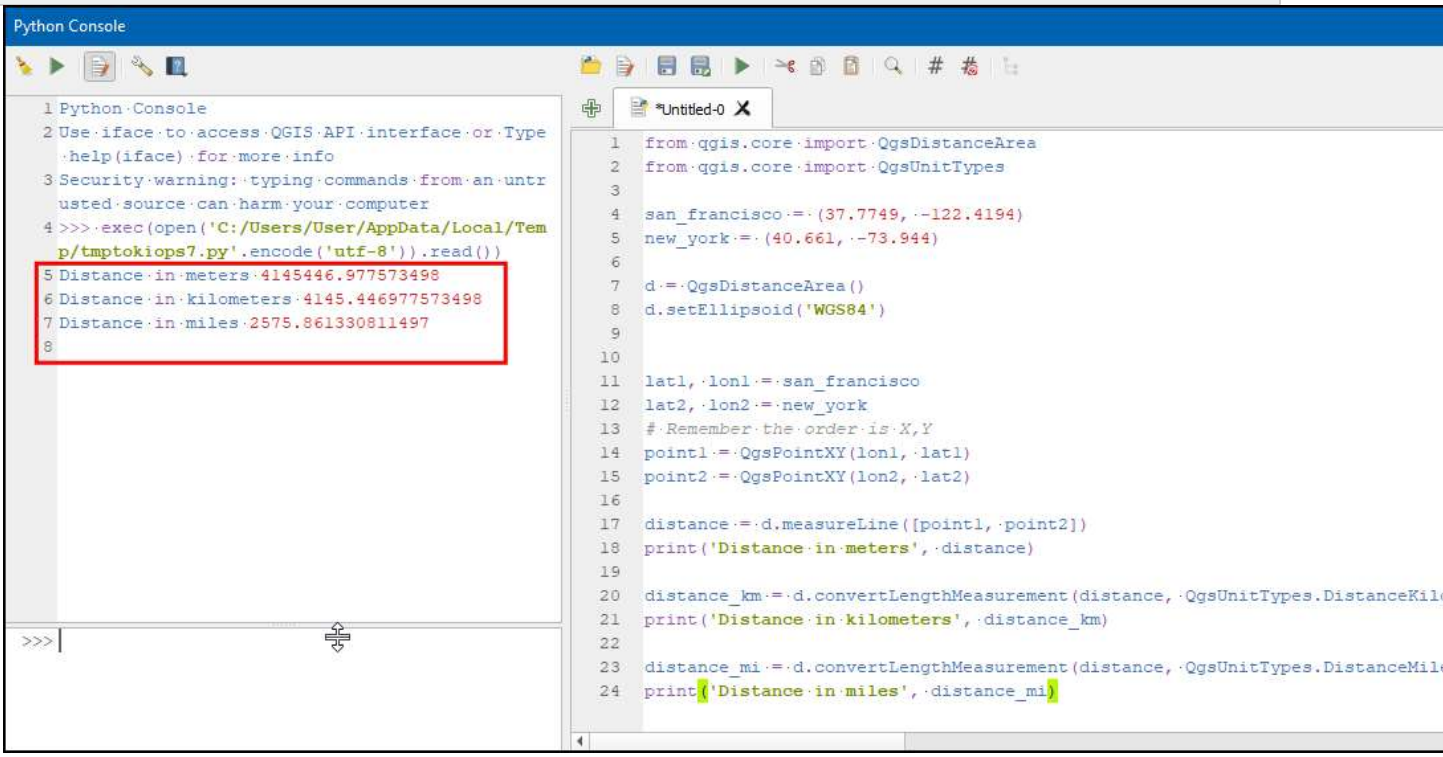
```
from qgis.core import QgsDistanceArea

san_francisco = (37.7749, -122.4194)
new_york = (40.661, -73.944)

d = QgsDistanceArea()
d.setEllipsoid('WGS84')
lat1, lon1 = san_francisco
lat2, lon2 = new_york
# Remember the order is X,Y
point1 = QgsPointXY(lon1, lat1)
point2 = QgsPointXY(lon2, lat2)

distance = d.measureLine([point1, point2])
print(distance/1000)
```

4.2 Distance Conversion

The distance returned by the measureLine() method in the previous section was in meters, and we divided it by 1000 to convert it to kilometers. Rather than doing this conversion manually, we can use the PyQGIS API. The QgsDistanceArea class has a method convertLengthMeasurement() that can convert the measured distance to any supported unit. The convertLengthMeasurement() method takes 2 arguments - the length measured by measureLine() method and the unit to convert the measurement to. The unit should be a value of the type QgsUnitTypes.DistanceUnit. The permitted values are defined in the QgsUnitType documentation. The code below shows how to convert the measured distance to Kilometers and Miles.

```
from qgis.core import QgsDistanceArea
from qgis.core import QgsUnitTypes

san_francisco = (37.7749, -122.4194)
new_york = (40.661, -73.944)

d = QgsDistanceArea()
d.setEllipsoid('WGS84')
lat1, lon1 = san_francisco
lat2, lon2 = new_york
# Remember the order is X,Y
point1 = QgsPointXY(lon1, lat1)
point2 = QgsPointXY(lon2, lat2)

distance = d.measureLine([point1, point2])
print('Distance in meters', distance)

distance_km = d.convertLengthMeasurement(distance, QgsUnitTypes.DistanceKilometers)
```

```
print('Distance in kilometers', distance_km)

distance_mi = d.convertLengthMeasurement(distance, QgsUnitTypes.DistanceMiles)
print('Distance in miles', distance_mi)
```



5. Graphical User Interface (GUI) Programming Basics

5.1 Qt and PyQt

Qt is a free and open-source widget toolkit for creating graphical user interfaces as well as cross-platform applications. QGIS is built using the Qt platform. Both Qt and QGIS itself have well-documented APIs that should be used when writing Python code to be run within QGIS.

PyQt is the Python interface to Qt. PyQt provides classes and functions to interact with Qt widgets.

5.2 Building a Dialog Box

Let's learn how to use PyQt classes to create and interact with GUI elements. Here we will create a simple dialog box that prompts a user for confirmation. You can type the code in the **Editor** and click **Run Script**.

```
mb = QMessageBox()
```

The QMessageBox is a PyQt class for creating a dialog with buttons. To use the class, you create object by *instantiating* the class. Here mb is an object, which is an instance of the QMessageBox class, created using the default parameters.

type() tells you what is the class of the object

```
type(mb)
```

dir returns list of the attributes and methods of any object

```
dir(mb)
```

Classes have methods that provide functionality. You can run the class methods on instance objects. For the QMessageBox class, setText() method will add a text to the dialog.

```
mb = QMessageBox()
mb.setText('Click OK to confirm')
```

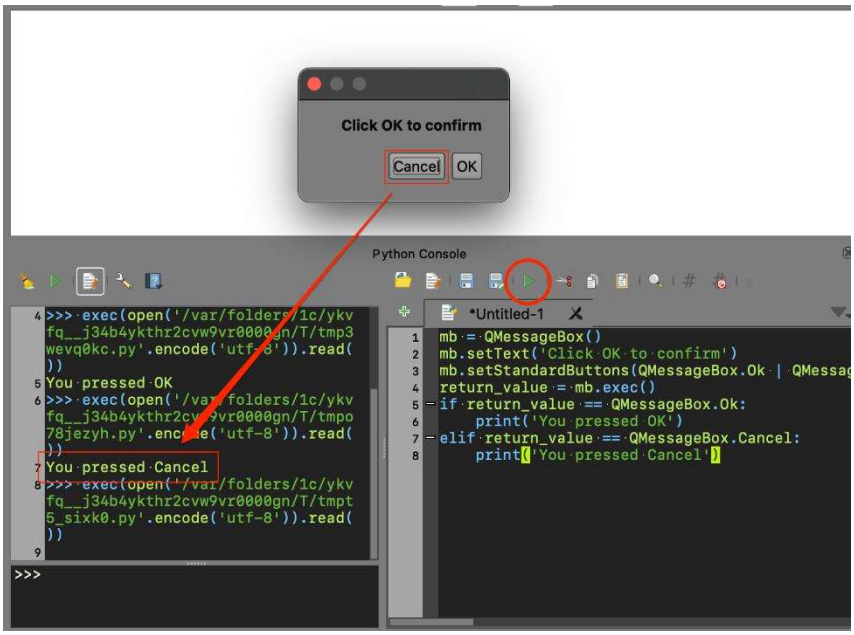
Classes also have class attributes which are shared across all instances. The QMessageBox class has Ok and Cancel attributes, which can be referred using QMessageBox.Ok and QMessageBox.Cancel.

```
mb = QMessageBox()
mb.setText('Click OK to confirm')
mb.setStandardButtons(QMessageBox.Ok | QMessageBox.Cancel)
```

To see the dialog, we need to use the exec() method. The user input is then captured and saved in the return_value variable.

The complete code snippet is as follows.Try it out and see the result of your action reflect in the Python Console.

```
mb = QMessageBox()
mb.setText('Click OK to confirm')
mb.setStandardButtons(QMessageBox.Ok | QMessageBox.Cancel)
return_value = mb.exec()
if return_value == QMessageBox.Ok:
    print('You pressed OK')
elif return_value == QMessageBox.Cancel:
    print('You pressed Cancel')
```



6.Deep Dive into PyQGIS

PyQGIS is the Python interface to QGIS. It is created using SIP and integrates with PyQt.

Fun Fact: Most QGIS class names start with the prefix *Qgs*. **Q** is for Qt and **gs** stands for Gary Sherman - the founder of the QGIS project.

QGIS C++ API documentation is available at https://qgis.org/api/

QGIS Python API documentation is available at https://qgis.org/pyqgis/3.0

Both C++ and Python APIs are identical for most part, but certain functions are not available in the Python API. [1]

6.1 QGIS Interface API (QgisInterface)

You are ready to dive into the PyQGIS API now. In this section, we will focus on the QgisInterface class - which provides methods for interaction with the QGIS environment. When QGIS is running, a variable called iface is set up to provide an object of the class QgisInterface to interact with the running QGIS environment. This interface allows access to the map canvas, menus, toolbars and other parts of the QGIS application. Python Console and Plugins can use iface to access various parts of the QGIS interface.

6.1.1 Change Title of QGIS Main Window

```
title = iface.mainWindow().windowTitle()
new_title = title.replace('QGIS', 'My QGIS')
iface.mainWindow().setWindowTitle(new_title)
```

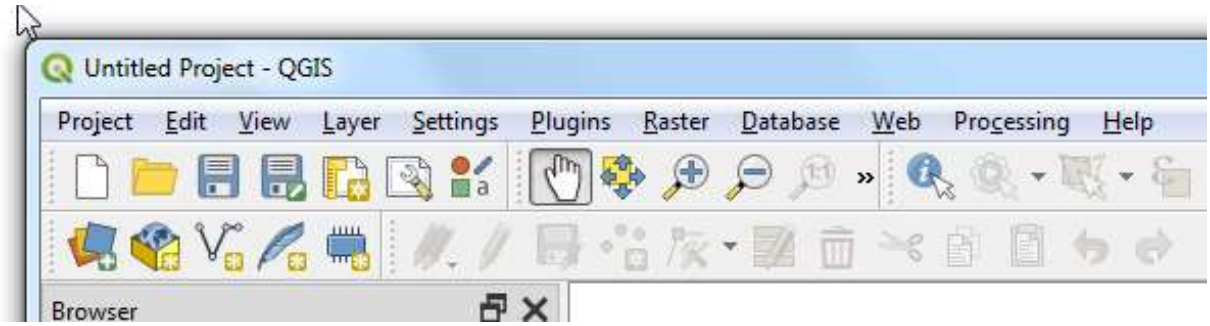
6.1.2 Change Icon of QGIS Main Window

os.path.expanduser('~') returns the path to the home directory of the user.

```
import os

icon = 'qgis-black.png'
data_dir = os.path.join(os.path.expanduser('~'), 'Downloads', 'pyqgis_in_a_day')
icon_path = os.path.join(data_dir, icon)
icon = QIcon(icon_path)
iface.mainWindow().setWindowIcon(icon)
```

6.1.3 Remove Raster and Vector Menus

```
vector_menu = iface.vectorMenu()
raster_menu = iface.rasterMenu()
menubar = vector_menu.parentWidget()
menubar.removeAction(vector_menu.menuAction())
menubar.removeAction(raster_menu.menuAction())
```



6.1.4 Understanding Signals and Slots

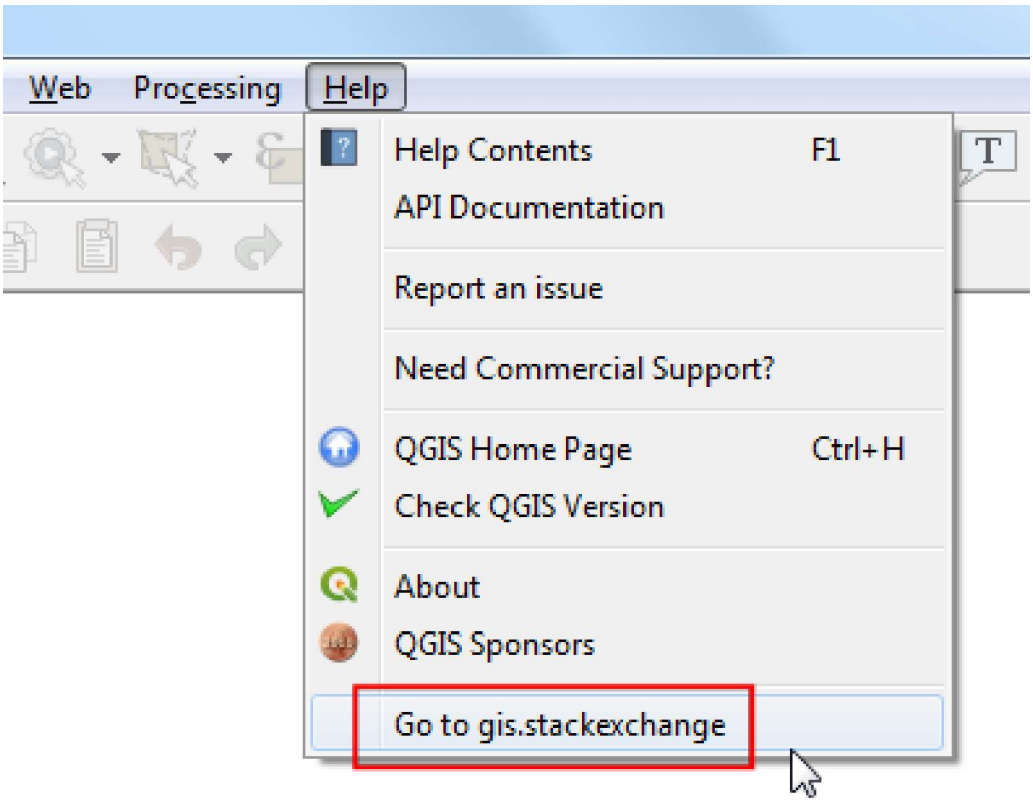
GUI programming requires responding to user's actions. All objects in Qt have a mechanism where they can emit a signal when there is a change in status. i.e. when a user *clicks* a button, or a window is *closed*. As a programmer, you can connect the signal to a slot (i.e. a python function) which will be called when the signal is emitted. The general syntax for connecting the signal to a slot is as follows

```
object.signal.connect(function)
```

6.1.5 Add A New Menu Item

A new button or menu item is created using QAction(). Here we create an action and then connect the *click* signal to a method that opens a website. [2]

```
import webbrowser

def open_website():
    webbrowser.open('https://gis.stackexchange.com')

website_action = QAction('Go to gis.stackexchange')
website_action.triggered.connect(open_website)
iface.helpMenu().addSeparator()
iface.helpMenu().addAction(website_action)
```



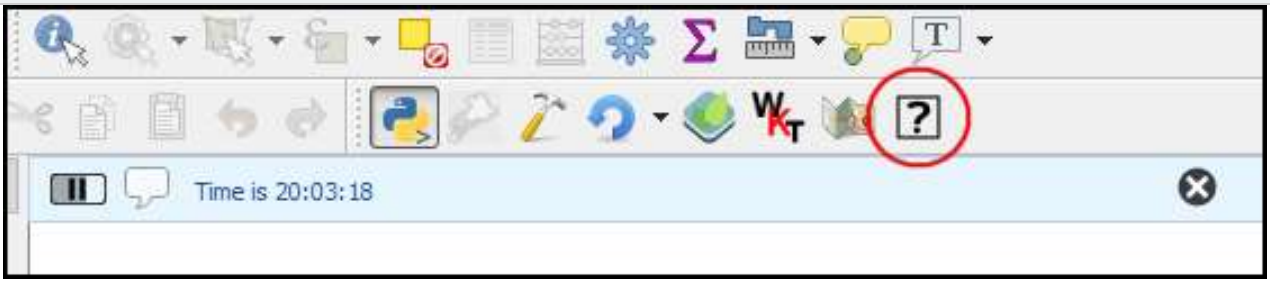
6.1.6 Change Visibility of a Toolbar

```
iface.pluginToolBar().setVisible(True)
```

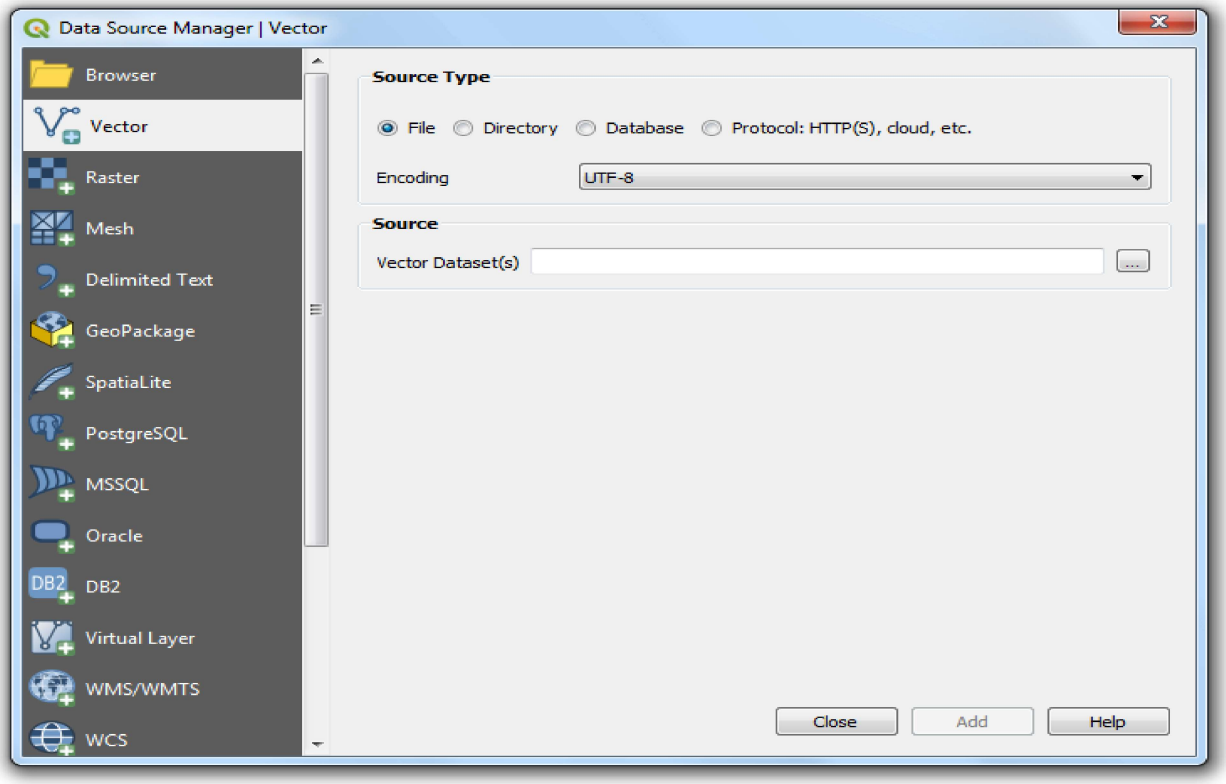
6.1.7 Add a button to a toolbar

```
import os
from datetime import datetime

icon = 'question.svg'
data_dir = os.path.join(os.path.expanduser('~'), 'Downloads', 'pyqgis_in_a_day')
icon_path = os.path.join(data_dir, icon)

def show_time():
    now = datetime.now()
```

```
    current_time = now.strftime("%H:%M:%S")
    iface.messageBar().pushInfo('Current Time', current_time)

action = QAction('Show Time')
action.triggered.connect(show_time)
action.setIcon(QIcon(icon_path))
iface.addToolBarIcon(action)
```



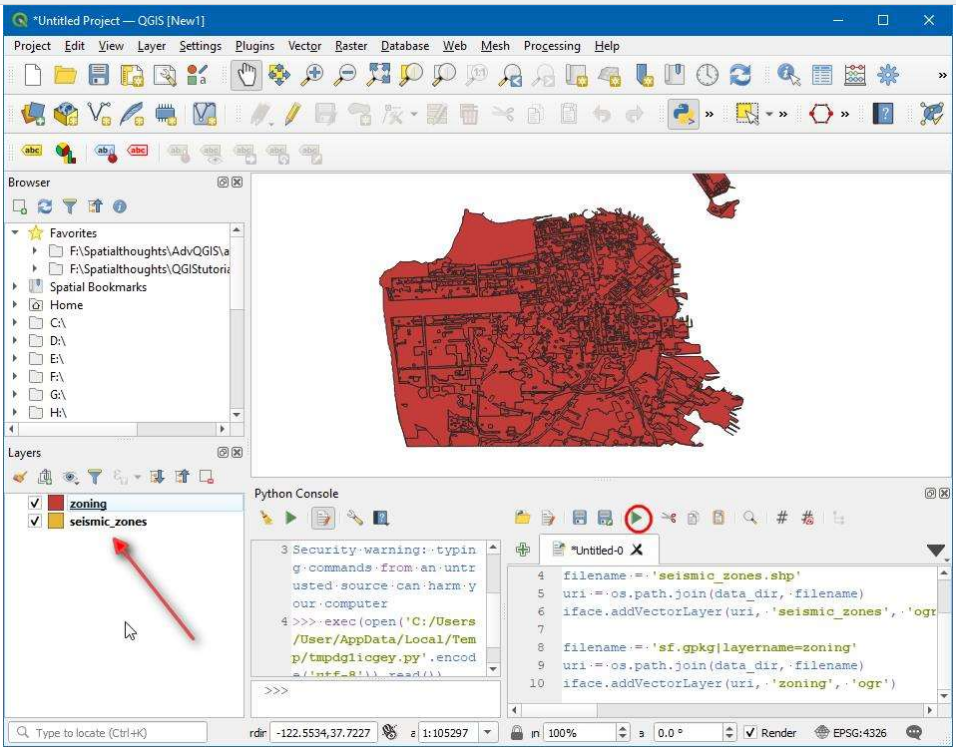
6.1.8 Add New Layers



Data sources are identified by an URI (Uniform Resource Identifier) - For files on computer the URI is the file path - For databases, the URI is constructed using the QgsDataSourceUri class and encodes,the database path, table, username, password etc. - For web layers, such as WMF/WFS etc, the URI is the web URL

```
import os
data_dir = os.path.join(os.path.expanduser('~'), 'Downloads', 'pyqgis_in_a_day')
```

```
filename = 'seismic_zones.shp'
uri = os.path.join(data_dir, filename)
iface.addVectorLayer(uri, 'seismic_zones', 'ogr')

filename = 'sf.gpkg|layername=zoning'
uri = os.path.join(data_dir, filename)
iface.addVectorLayer(uri, 'zoning', 'ogr')
```



6.1.9 Change name of a Layer

```
layer = iface.activeLayer()
name = layer.name()
layer.setName('sf_' + name)
```

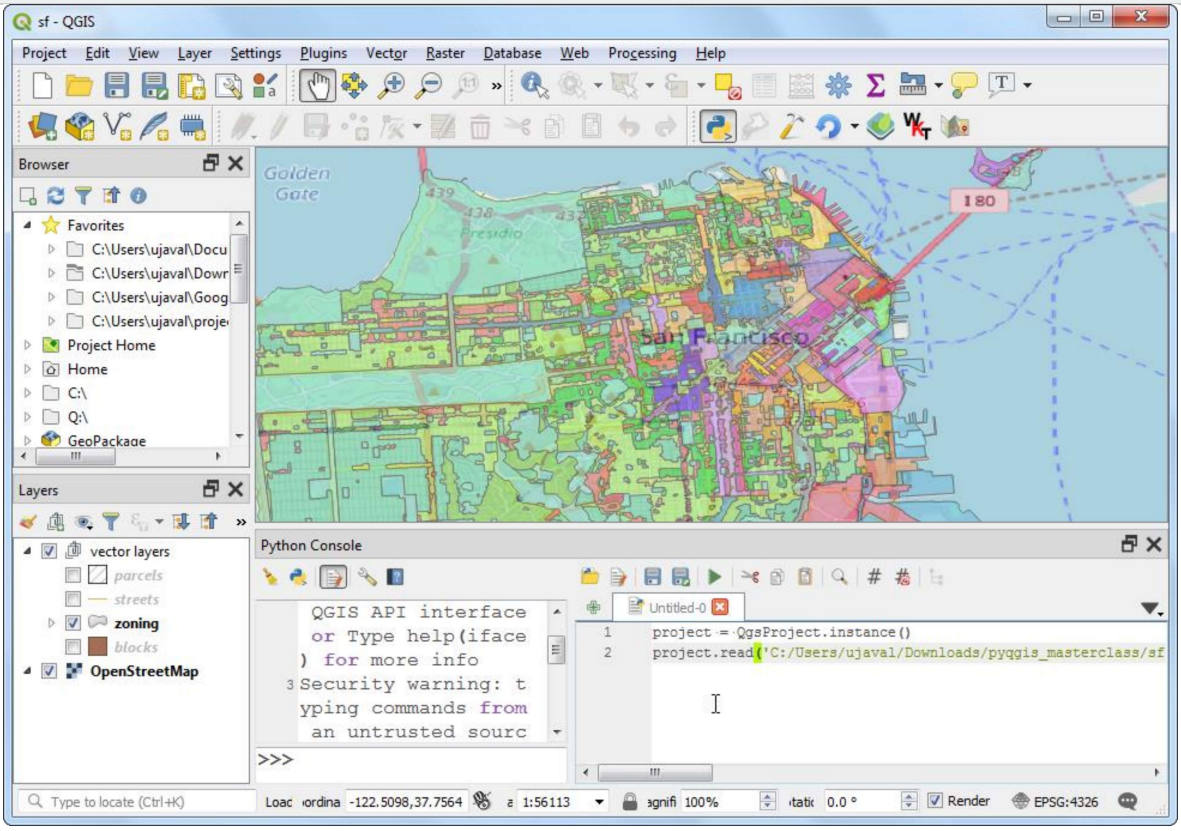
6.2 QGIS Project API (QgsProject)

Another very important QGIS class is QgsProject. This class is used for all operations in a QGIS project - including adding/removing map layers, styling, print layouts etc. The QgsProject is a **Singleton Class** - meaning it can have only 1 instance at a time. The instance refers to the current QGIS project that is loaded. When QGIS starts, a blank project is created. When you load another project, the existing project is closed and a new project instance is created. You can get the current instance of the QgsProject class by calling the **QgsProject.instance()** method.

6.2.1 Load a project

```
import os
data_dir = os.path.join(os.path.expanduser('~'), 'Downloads', 'pyqgis_in_a_day')
```

```
project = QgsProject.instance()
project_name = 'sf.qgz'
project_path = os.path.join(data_dir, project_name)
project.read(project_path)
```



6.2.2 Search for a layer

```
blocks = QgsProject.instance().mapLayersByName('blocks')[0]
```

7. Running Python Code at QGIS Launch

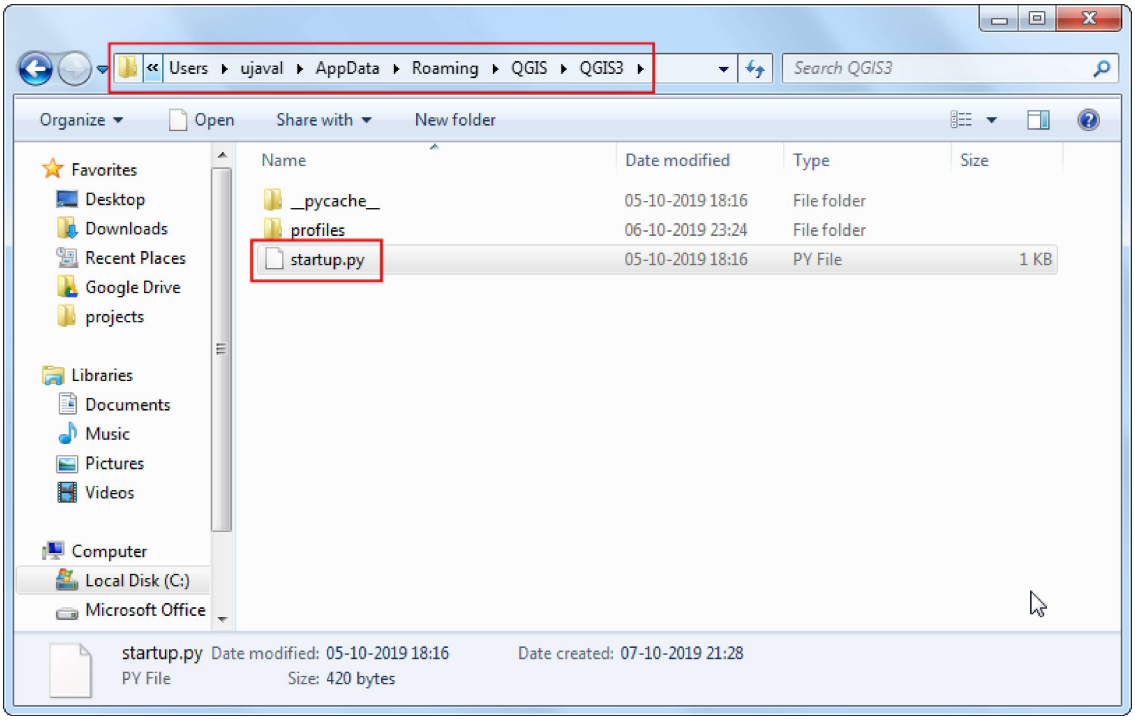
It is possible to execute some PyQGIS code every time QGIS starts. QGIS looks for a file named startup.py in the user's Python home directory, and if it is found, executes it. This file is very useful in customizing QGIS interface with techniques learnt in the previous section.

If you are running multiple versions of QGIS, a very useful customization is to display the QGIS version number and name in the main window. The version name is stored in a global QGIS variable called qgis_version. We can read that variable and set the main window's title with it. We connect this code to the signal iface.initializationCompleted signal when the main window is loaded.

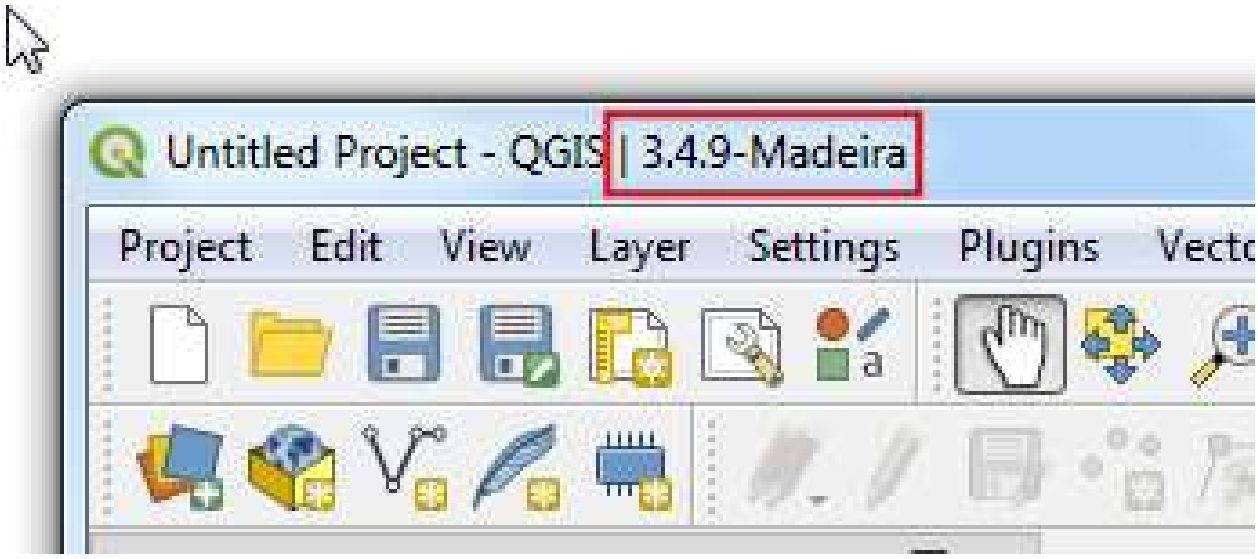
Create a new file named startup.py with the following code. Note the imports at the top - including iface. When we ran the code snippets in the Python Console, we did not have to import any modules since they are done automatically when the console starts. For pyqgis scripts elsewhere, we have to explicitly import the modules (classes) that we want to use.

```
from qgis.utils import iface
from qgis.core import QgsExpressionContextUtils

def customize():
    version = QgsExpressionContextUtils.globalScope().variable('qgis_version')
    title = iface.mainWindow().windowTitle()
    iface.mainWindow().setWindowTitle('{} | {}'.format(title,version))


iface.initializationCompleted.connect(customize)
```

This file needs to be copied to the appropriate directory on your system. See QGIS documentation for details on the path for your platform.



Once you copy the file at that location, restart QGIS. The title bar should now have the QGIS version name in it.



Pro Tip: It is possible to put the startup.py file on a shared drive for enterprise deployment of QGIS customizations. Learn more.

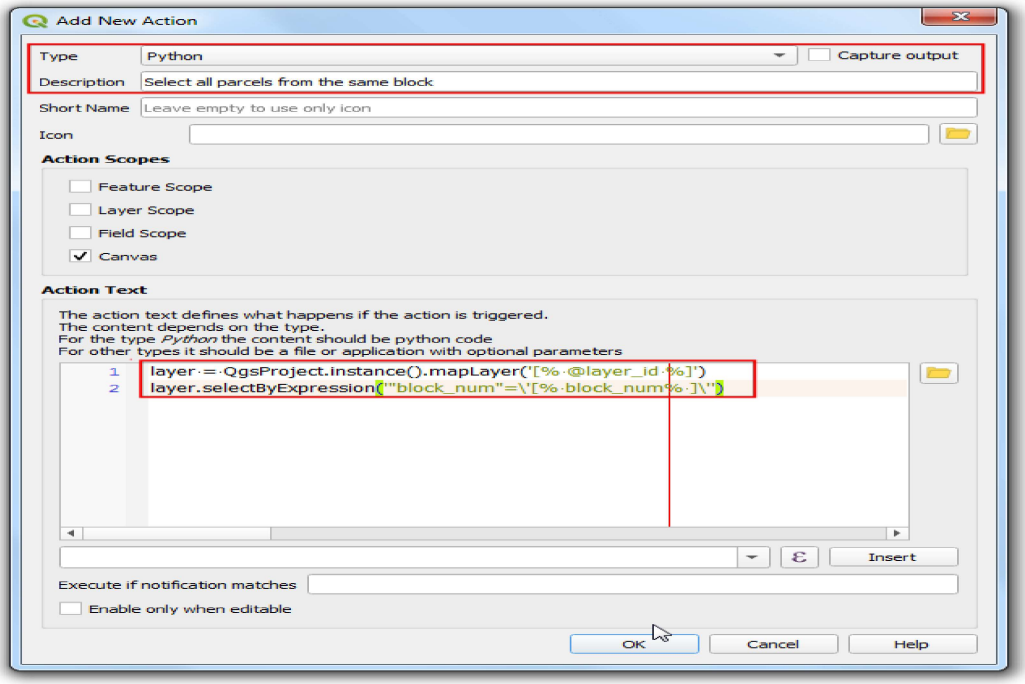
8. Creating Custom Python Actions

Actions in QGIS provide a quick and easy way to trigger custom behavior in response to a user's action - such as click on a feature in the canvas or an attribute value in the attribute table.

Actions provide an easy way to add custom behavior to QGIS without having to write plugins. Actions are integrated into the QGIS GUI and allow you to execute PyQGIS code on vector layers.

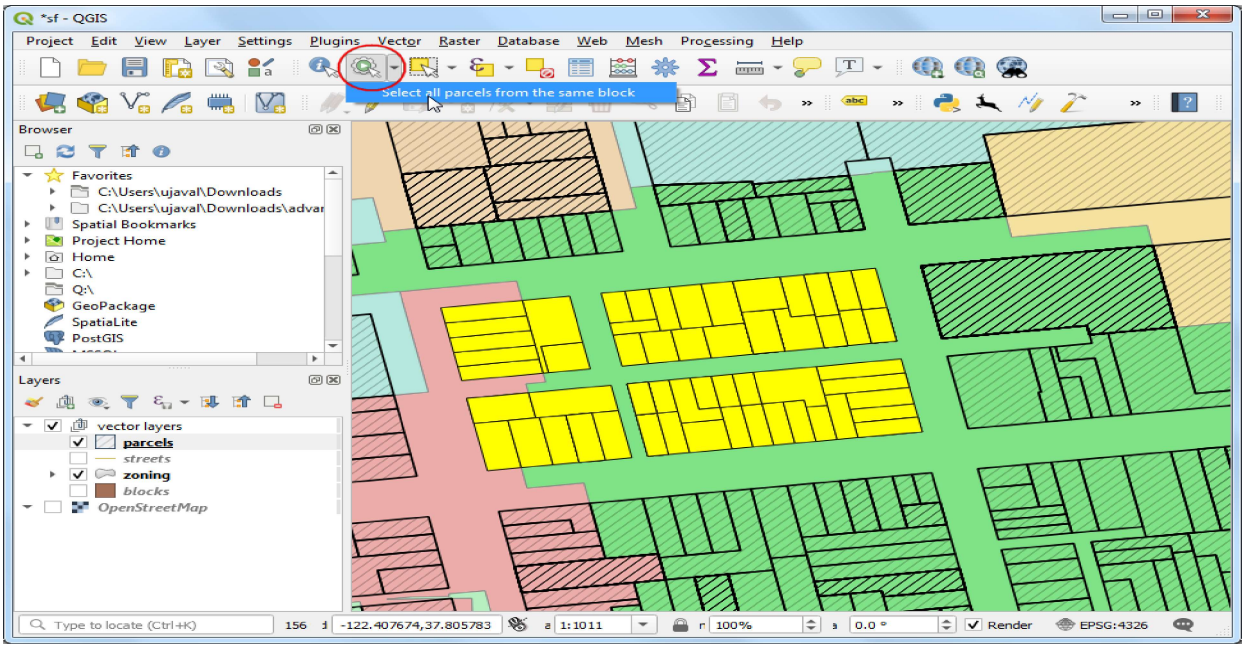
Actions are defined at the layer level. We will define an action for the parcels layer so when a user clicks on a parcel, all parcels that belong to the same block will be selected. The parcel layer has an attribute **block_num** that refers to the block that parcel belongs to.

Load the sf.qgz project from your data package. Right-click the parcels layer, click *Properties* and switch to the *Actions* tab. Click *Add a new action* button. Select **Python** as the *Type*. Name the action as **Select all parcels from the same block**. This action is meant to be used for selecting features on the map canvas, so check the **Canvas** as the *Action Scope*. Enter the following code snippet in the *Action Text* and click *OK*. [3]



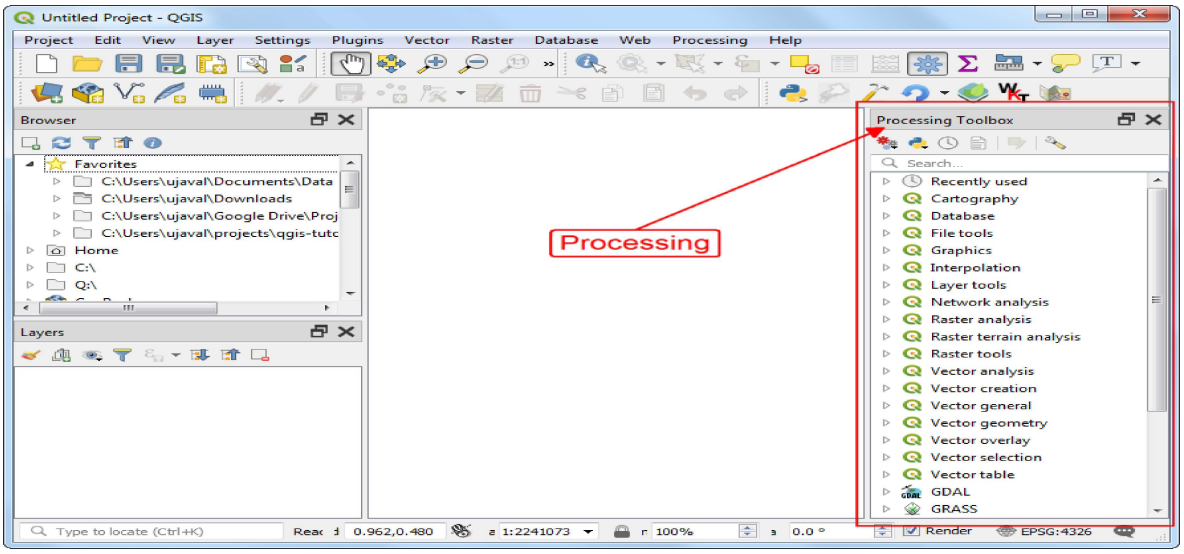

```
layer = QgsProject.instance().mapLayer('[% @layer_id %]')
layer.selectByExpression('"block_num"=\'[% block_num %]\'')
```

Back in the main QGIS window, enable the parcels layer. Click the *Run feature action* button from the *Attributes Toolbar* and select **Select all parcels from the same block**. With the action enabled, click on any parcel and you will see all parcels from the block get selected.



9. Running Processing Algorithms

While the PyQGIS API providers many functions to work with layers, features, attributes and geometry - it is a much better practice to use the built-in processing algorithms to alter the layers or do any analysis. This will give you better performance and result in much lesser code. Here are some examples on how to use processing algorithms from Python to do vector and raster layer editing. You will find more information about various options and techniques in the article Using processing algorithms from the console of the QGIS User Guide

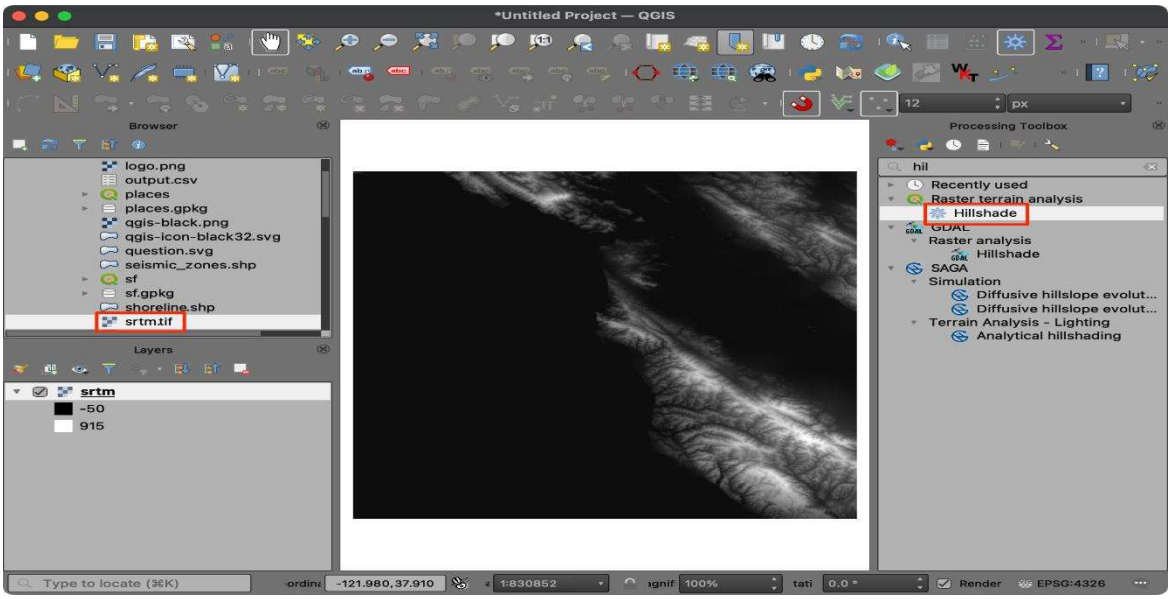


9.1 Creating Hillshade from a DEM

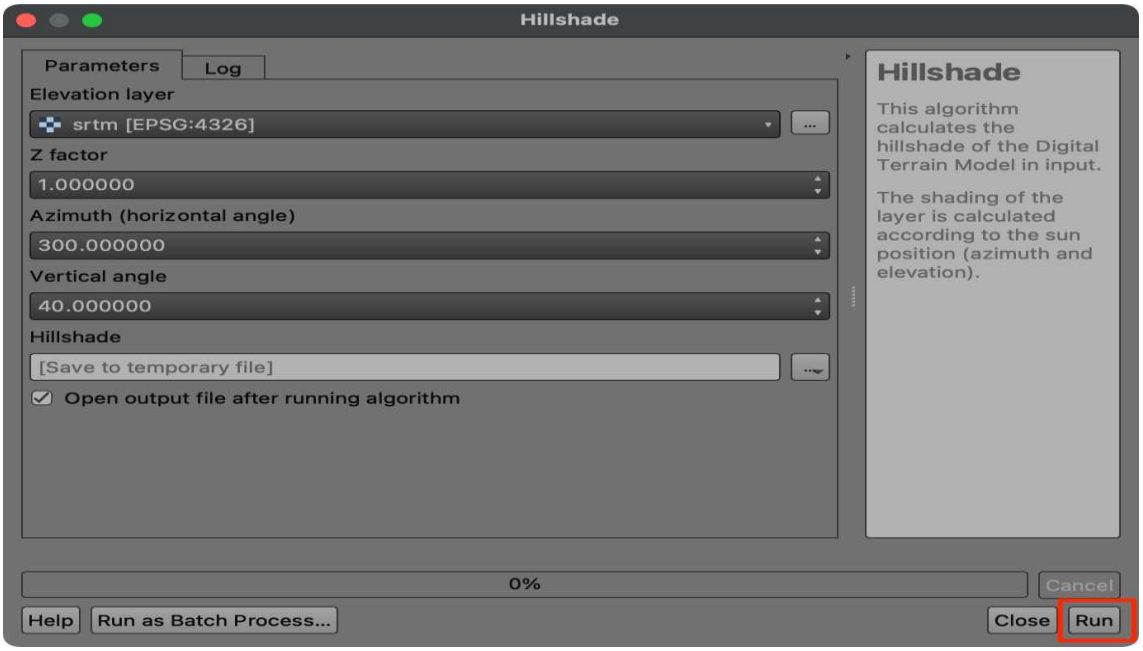
To use any Processing Algorithm via Python, you need to know how to specify all the required parameters. This is easiest to obtain by running the algorithm via the GUI first.

In this section we will learn how to create a hillshade raster from a DEM. We will first carry out the task using QGIS.

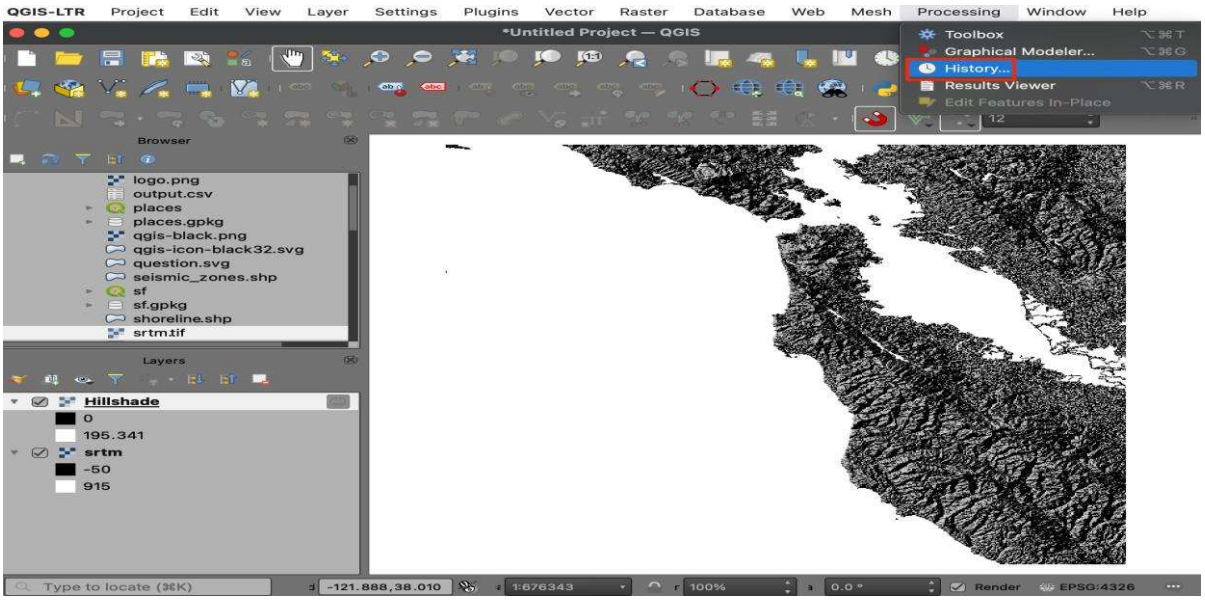
1. Browse to the data directory and load the srtm.tif layer. Search and locate the **Processing Toolbox → Raster terrain analysis → Hillshade** algorithm from the *Processing Toolbox*. Double-click to open it.



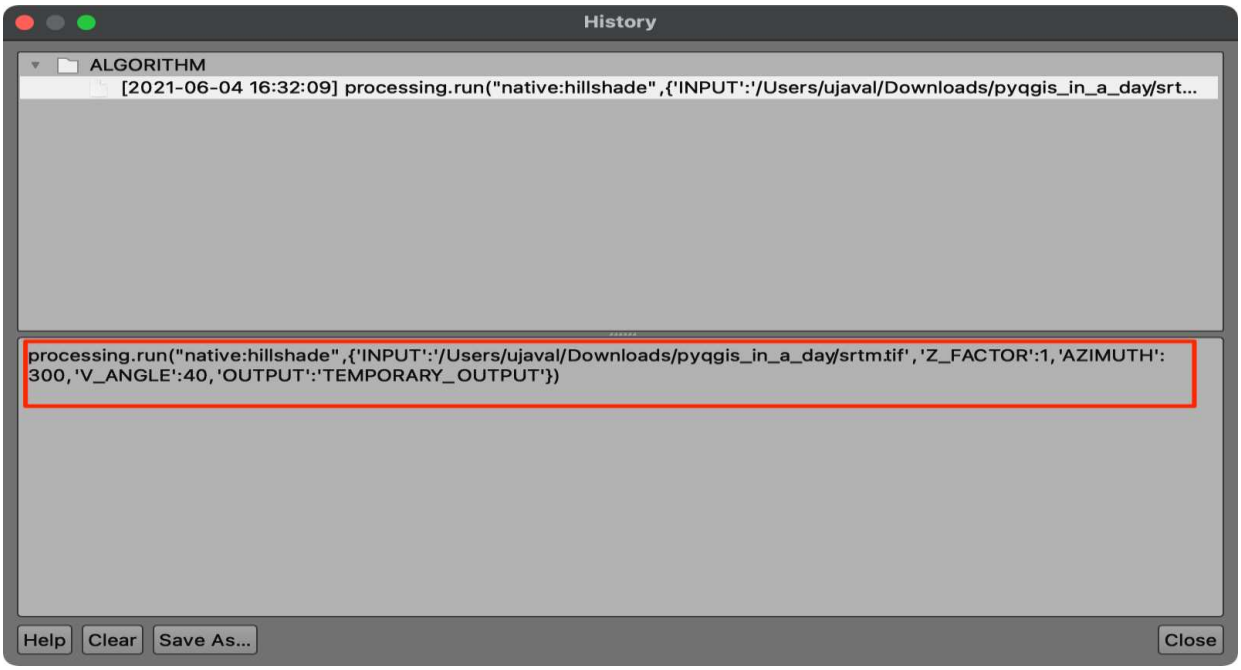
2. Select srtm as the *Elevation layer* and keep all the other parameters to their default value. Click *Run*.



3. A new Hillshade layer will be added to the *Layers* panel. Now we will locate the Python command for this operation from the *Processing History*. Go to **Processing → History*.



4. The first entry in the top panel will show the last algorithm that was ran from the toolbox. Click on it to select it. The full Python command will be shown at the bottom. Copy it.



You can now use the parameters in your Python code and replace the path of the input. Below is the code snippet that runs the same algorithm from a Python script. Note that we are using the processing.runAndLoadResults() method that adds the resulting layer to the canvas.

```
import os
data_dir = os.path.join(os.path.expanduser('~'), 'Downloads', 'pyqgis_in_a_day')

filename = 'srtm.tif'
srtm = os.path.join(data_dir, filename)
```

```
iface.addRasterLayer(srtm, 'srtm', 'gdal')

results = processing.runAndLoadResults("native:hillshade",
   {'INPUT': srtm,
   'Z_FACTOR':2,
   'AZIMUTH':300,
   'V_ANGLE':40,
   'OUTPUT': 'TEMPORARY_OUTPUT'})
```

9.2 Running Multiple Processing Algorithms

We can also *chain* multiple processing tools to build a script to build a data processing pipeline. In the example below, we will do 2 steps

1. Clip srtm.tif raster using the shoreline.shp layer.
2. Calculate the hillshade on the clipped raster and load it in QGIS.

Note that we are using the processing.run() method for the first step. This method calculates the output, but does not load the result to QGIS. This allows us to carry out multiple processing steps and not load intermediate layers.

```
import os
data_dir = os.path.join(os.path.expanduser('~'), 'Downloads', 'pyqgis_in_a_day')

filename = 'srtm.tif'
srtm = os.path.join(data_dir, filename)
filename = 'shoreline.shp'
shoreline = os.path.join(data_dir, filename)

results = processing.run("gdal:cliprasterbymasklayer",
   {'INPUT':srtm,
   'MASK': shoreline,
   'OUTPUT':'TEMPORARY_OUTPUT'})

clipped_dem = results['OUTPUT']

results = processing.runAndLoadResults("native:hillshade",
   {'INPUT': clipped_dem,
   'Z_FACTOR':2,
   'AZIMUTH':300,
   'V_ANGLE':40,
   'OUTPUT': 'TEMPORARY_OUTPUT'})
```

You can also do batch-processing by iterating through multiple layers and running the processing algorithm in a for-loop. Doing it via Python allows you greater flexibility - such as combining the results into a single layer.

10. Advanced Python Concepts

10.1 Understanding Python Iterators

An *Iterator* is a type of Python object that contains items that can be iterator upon. They are similar to other objects, like *lists* - but with a key difference. When you create an iterator, you don't store all the items in memory. The iterator loads a single item at a time and then fetches the next item when asked for it. This makes it very efficient for reading large amounts of data without having to read the entire dataset. QGIS implements iterators for many different object types.

We will continue to work with the sf.qgz project. Open the project and select the blocks layer. In the example below, the result of calling layer.getFeatures() is an iterator. You can call the next() function to fetch the next item from the iterator.

```
layer = iface.activeLayer()
features = layer.getFeatures()
f = next(features)
print(f.attributes())
f = next(features)
print(f.attributes())
```

You can also use for-loops to iterate through an iterator. Here we look up the *Feature ID* of each feature using the id() method and store it in a list.

```
layer = iface.activeLayer()
features = layer.getFeatures()
ids = []
for f in features:
  id = f.id()
  ids.append(id)

print(ids)
```

10.2 List Comprehensions

A common data processing task in Python is to read items from a list or an iterator, doing some processing on each item and creating a new list with the results. The regular way to do this is to first create an empty list, iterate over each item of the existing list, and append the results to the new empty list. Python provides a powerful alternative to this workflow in the form of *List Comprehension*. The snippet below shows the syntax.

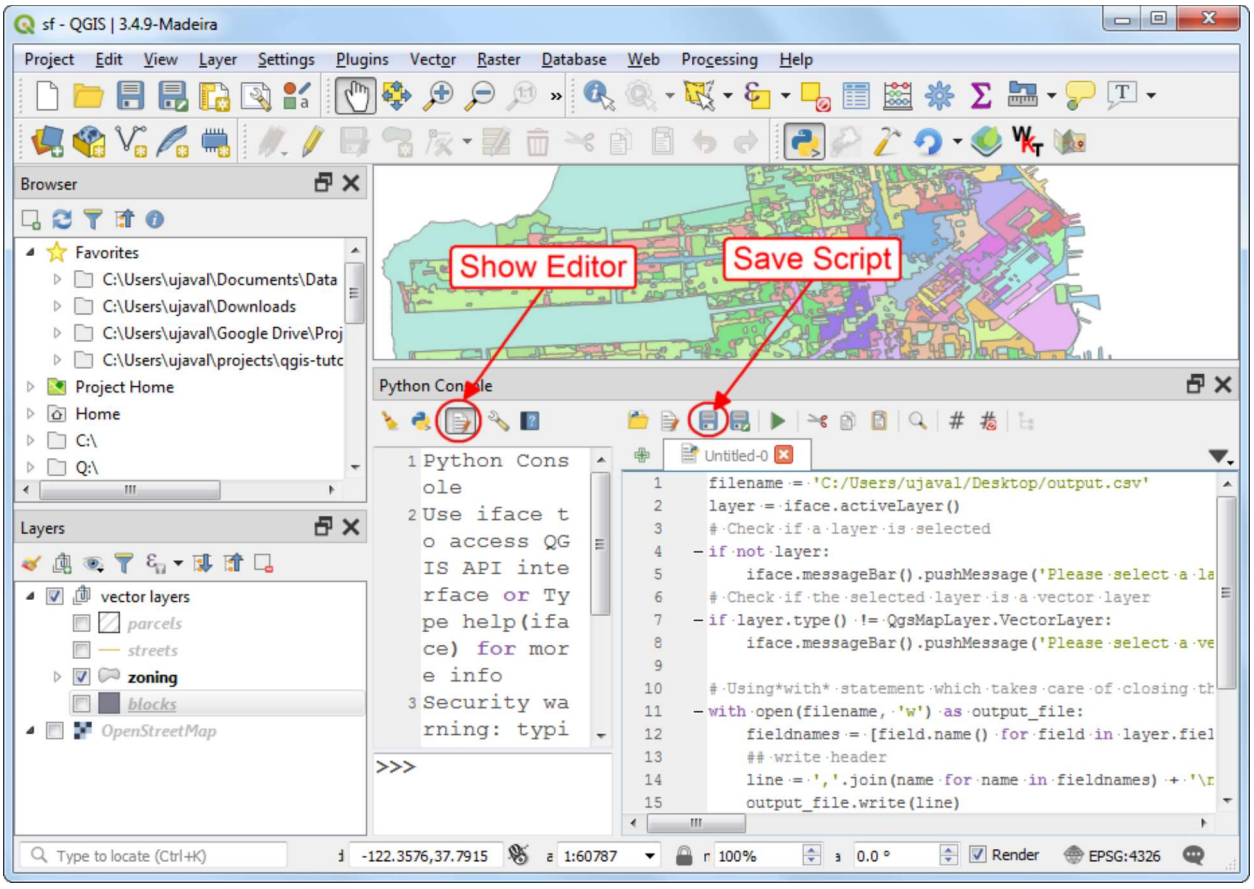
```
my_list = [1, 2, 3, 4, 5]

# Create a new list by adding 1 to each element
new_list = []
for x in my_list:
   new_list.append(x+1)

print(new_list)

# Use list comprehension syntax

new_list = [x+1 for x in my_list]
print(new_list)
```

11. Writing Python Console Scripts

We will now see how can we write python scripts, save them and run them within QGIS. We will also see examples of best practices - such as validating inputs, writing files safely and communicating the status to the user.

A goal of the script is to read the features of a selected vector layer, and write its attributes to a CSV file.

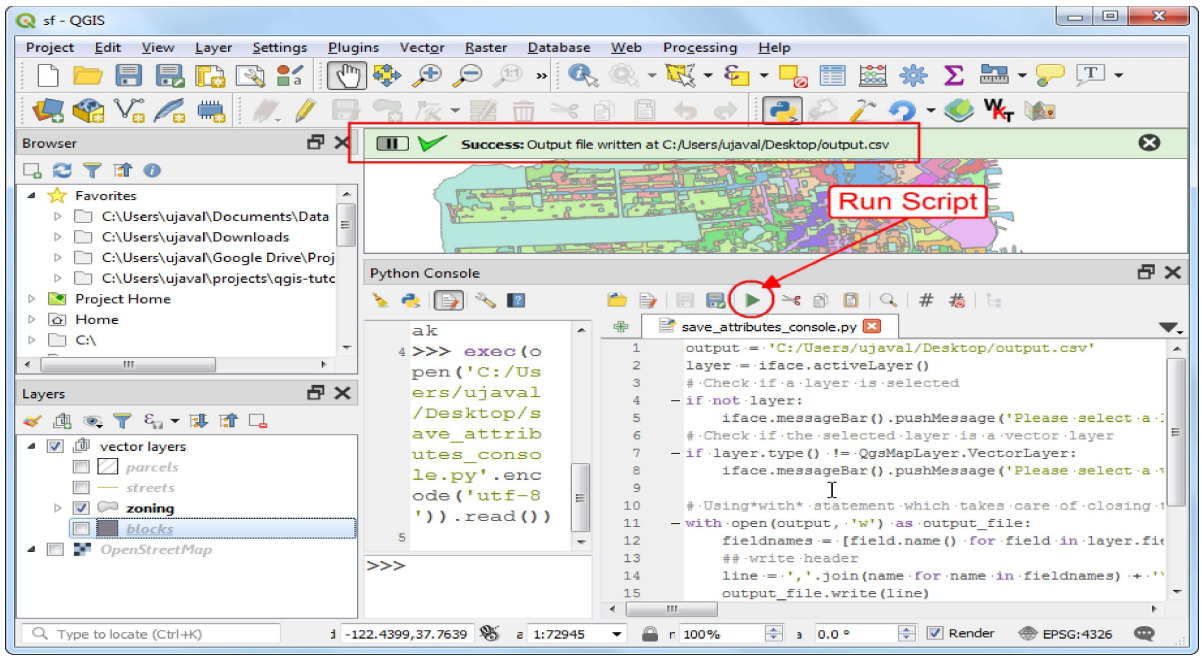
Open the Python Console and click the *Show Editor* button. Copy/paste the following code. Click the *Save* button and save the file as save_attributes_console.py. The file can be saved anywhere.




```
import os
data_dir = os.path.join(os.path.expanduser('~'), 'Downloads', 'pyqgis_in_a_day')

output_name = 'output.csv'
output_path = os.path.join(data_dir, output_name)

layer = iface.activeLayer()
# Check if a layer is selected
```

```
if not layer:
    iface.messageBar().pushMessage('Please select a layer',  level=Qgis.Critical)
# Check if the selected layer is a vector layer
if layer.type() != QgsMapLayer.VectorLayer:
    iface.messageBar().pushMessage('Please select a vector layer',  level=Qgis.Critical)

# Using*with* statement which takes care of closing the files and handling errors
with open(output_path, 'w') as output_file:
    fieldnames = [field.name() for field in layer.fields()]
    ## write header
    line = ','.join(name for name in fieldnames) + '\n'
    output_file.write(line)
    # write feature attributes
    for f in layer.getFeatures():
        line = ','.join(str(f[name]) for name in fieldnames) + '\n'
        output_file.write(line)
iface.messageBar().pushMessage(
    'Success:', 'Output file written at ' + output_path, level=Qgis.Success)
```

Select the blocks layer and run this script by clicking the *Run Script* button. The script will process the layer and write the file at the given location.



12. Processing Scripts

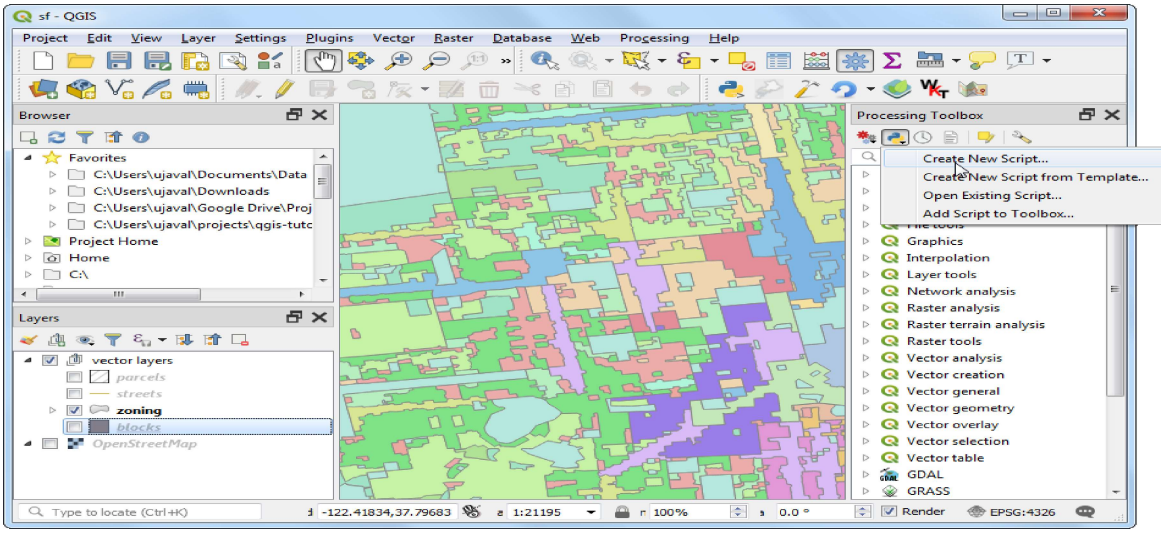
We saw how to write a Python script in the QGIS Python Console Code Editor. But there is another way - and it is the preferred approach to write scripts. Whenever you are writing a new script, consider using the built-in **Processing Framework**. This has several advantages. First, taking user input and writing output files is far easier because Processing Framework offers standardized user interface for these. Second, having your script in the Processing Toolbox also allows it to be part of

any Processing Model or be run as a Batch job with multiple inputs. This tutorial will show how to write a custom python script that can be part of the Processing Framework in QGIS.

We will now see how the Python Console script can be converted to a Processing script.

12.2 Writing a Processing Script

To create a new processing script, go to **Processing → Processing Toolbox**. Click the *Scripts* button and select *Create New Script...*

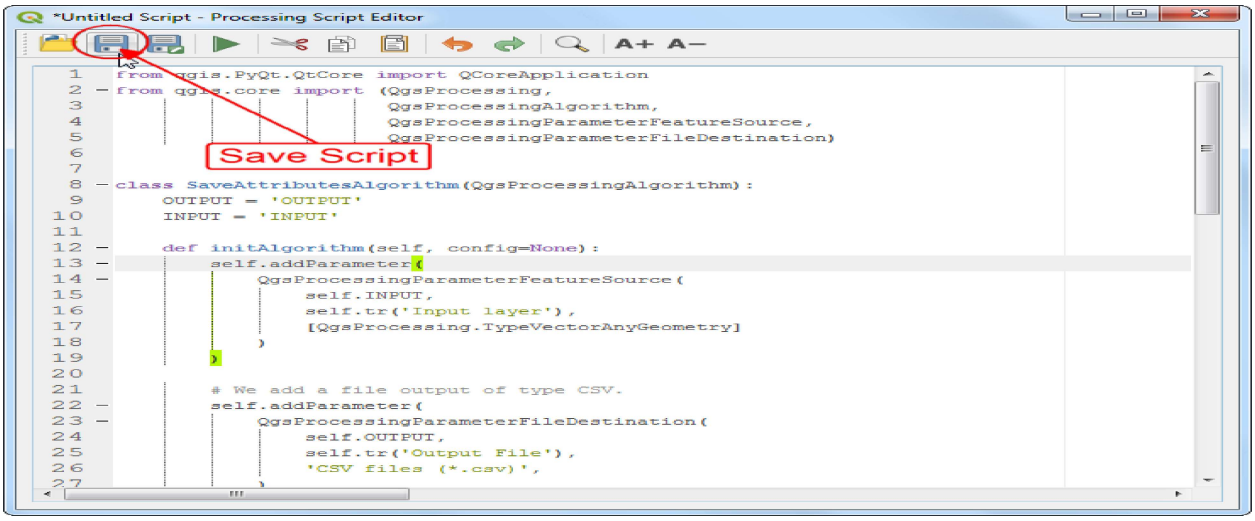


Copy/paste the following code into the *Processing Script Editor*.

save_attributes_processing.py

```
from PyQt5.QtCore import QCoreApplication
from qgis.core import (QgsProcessing,
                       QgsProcessingAlgorithm,
                       QgsProcessingParameterFeatureSource,
                       QgsProcessingParameterFileDestination)


class SaveAttributesAlgorithm(QgsProcessingAlgorithm):
    """Saves the attributes of a vector layer to a CSV file."""
    OUTPUT = 'OUTPUT'
    INPUT = 'INPUT'

    def initAlgorithm(self, config=None):
        self.addParameter(
            QgsProcessingParameterFeatureSource(
                self.INPUT,
                self.tr('Input layer'),
                [QgsProcessing.TypeVectorAnyGeometry]
            )
        )

        # We add a file output of type CSV.
        self.addParameter(
            QgsProcessingParameterFileDestination(
```

```
            self.OUTPUT,
            self.tr('Output File'),
            'CSV files (*.csv)',
        )
    )

def processAlgorithm(self, parameters, context, feedback):
    source = self.parameterAsSource(parameters, self.INPUT, context)
    csv = self.parameterAsFileOutput(parameters, self.OUTPUT, context)

    fieldnames = [field.name() for field in source.fields()]

    # Compute the number of steps to display within the progress bar and
    # get features from source
    total = 100.0 / source.featureCount() if source.featureCount() else 0
    features = source.getFeatures()

    with open(csv, 'w') as output_file:
      # write header
      line = ','.join(name for name in fieldnames) + '\n'
      output_file.write(line)
      for current, f in enumerate(features):
          # Stop the algorithm if cancel button has been clicked
          if feedback.isCanceled():
              break

          # Add a feature in the sink
          line = ','.join(str(f[name]) for name in fieldnames) + '\n'
          output_file.write(line)

          # Update the progress bar
          feedback.setProgress(int(current * total))

    return {self.OUTPUT: csv}

def name(self):
    return 'save_attributes'

def displayName(self):
    return self.tr('Save Attributes As CSV')

def group(self):
    return self.tr(self.groupId())

def groupId(self):
    return ''

def tr(self, string):
    return QCoreApplication.translate('Processing', string)

def createInstance(self):
    return SaveAttributesAlgorithm()
```

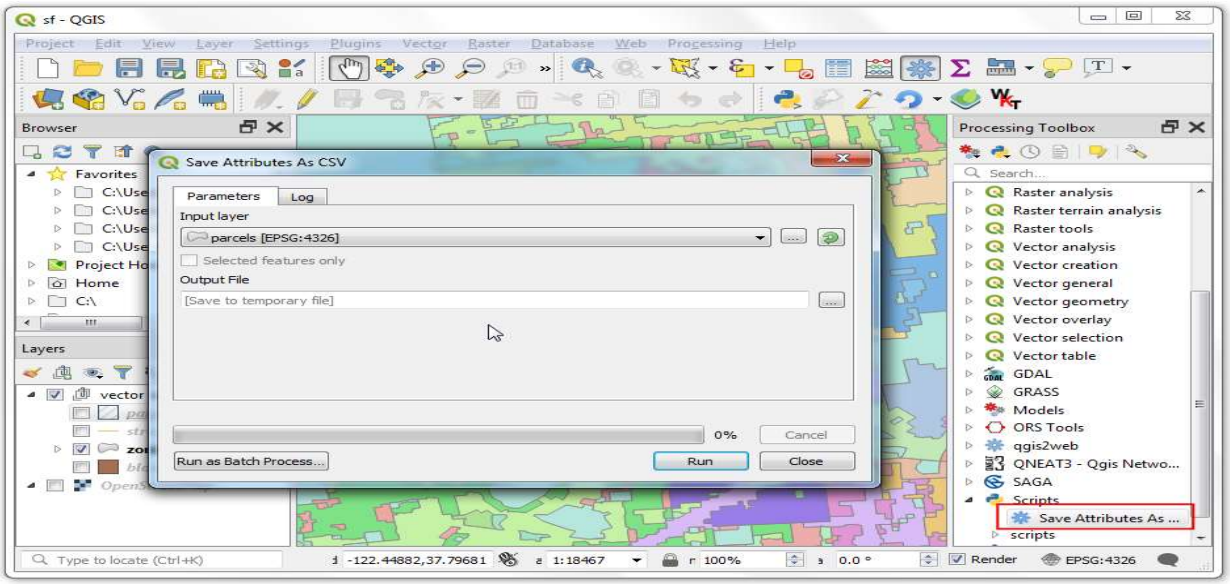


By now, you must be familiar with the code in the processAlgorithm() method. This method contains the main logic of the script that gets executed when the user clicks the *Run* button. Here we are creating a new class called SaveAttributesAlgorithm. As this is a processing script, we must inherit from the QgsProcessingAlgorithm class. The initAlgorithm() method is called to set-up the inputs and the outputs. There are other methods that set the algorithm name and group.

Click the *Save Script* button and save the script as save_attributes_algorithm.py. This script must be saved inside the {profile foler}/processing/scripts/ directory so it can be loaded when QGIS starts.

Once saved, the algorithm will appear in the *Processing Toolbox* under **Scripts → Save Attributes As CSV**. Double-click to launch it. You will see the standard processing algorithm dialog where the user can select inputs and outputs easily. Progress bar is shown correctly and the execution also stops if the user presses the *Cancel* button. If the large is large and the algorithm would take time to process it, the user can also close the window and the algorithm will continue to run in the background.

You can review additional information and tips in the Writing new Processing algorithms as Python scripts section of the QGIS User Guide.



13. Writing Plugins

Plugins are a great way to extend the functionality of QGIS. You can write plugins using Python that can range from adding a simple button to sophisticated tool-kits. There are 2 broad categories of python plugins - Processing Plugins and GUI Plugins. We will cover both in this section.

There is a plugin named Plugin Builder that can help you generate a starter plugin. I have published step-by-step instructions for both GUI plugins and Processing Plugins using the Plugin Builder method. While this method gives you an easy way to have a functional plugin, it is not the ideal way to learn plugin development. I prefer starting from a minimal template and adding elements as and when needed. Here we will learn the basics of plugin framework using a minimal plugin and learn how to add various element to make it a full plugin.

13.1 A Minimal Plugin

Plugins are much more integrated into the QGIS system than Python Scripts. They are managed by **Plugin Manager** and are initialized when QGIS starts. To understand the required structure, let's see what a minimal plugin looks like. You can learn more about this structure at QGIS Minimalist Plugin Skeleton.

We will be the same script for saving attributes of a vector layer and convert it to a plugin. The first requirement for plugins is a file called metadata.txt. This file contains general info, version, name and some other metadata used by plugins website and plugin manager.

metadata.txt

```
[general]
name=Save Attributes
description=This plugin saves the attributes of the selected vector layer as a CSV file
version=1.0
qgisMinimumVersion=3.0
author=Ujaval Gandhi
email=ujaval@spatialthoughts.com
```

Second is the file that contains the main logic of the plugin. It must have __init__() method that gives the plugin access to the QGIS Interface (iface). The initGui() method is called when the plugin is loaded and unload() method which is called when the plugin is unloaded. For now, we are creating a minimal plugin that just add a button and a menu entry that displays message when clicked.

save_attributes.py

```
import os
import sys
import inspect
from PyQt5.QtWidgets import QAction
from PyQt5.QtGui import QIcon

cmd_folder = os.path.split(inspect.getfile(inspect.currentframe()))[0]

class SaveAttributesPlugin:
    def __init__(self, iface):
```

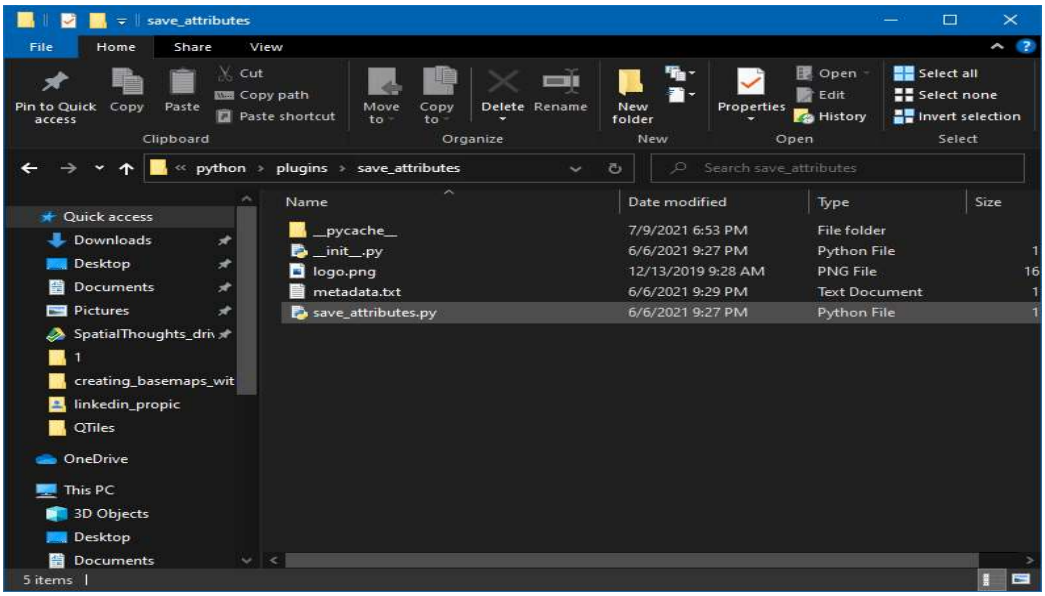
```
        self.iface = iface

    def initGui(self):
        icon = os.path.join(os.path.join(cmd_folder, 'logo.png'))
        self.action = QAction(QIcon(icon), 'Save Attributes as CSV', self.iface.mainWindow())
        self.action.triggered.connect(self.run)
        self.iface.addPluginToMenu('&Save Attributes', self.action)
        self.iface.addToolBarIcon(self.action)

    def unload(self):
        self.iface.removeToolBarIcon(self.action)
        self.iface.removePluginMenu('&Save Attributes', self.action)
        del self.action

    def run(self):
        self.iface.messageBar().pushMessage('Hello from Plugin')
```

Third file is called __init__.py which is the starting point of the plugin. It imports the plugin class created in the second file and creates an instance of it.

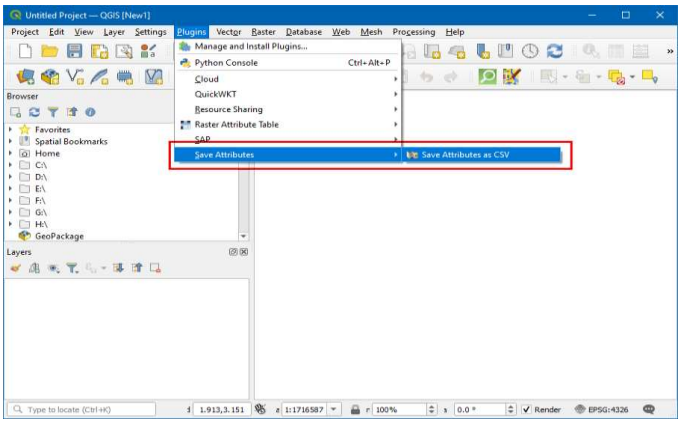
__init__.py

```
from .save_attributes import SaveAttributesPlugin

def classFactory(iface):
    return SaveAttributesPlugin(iface)
```

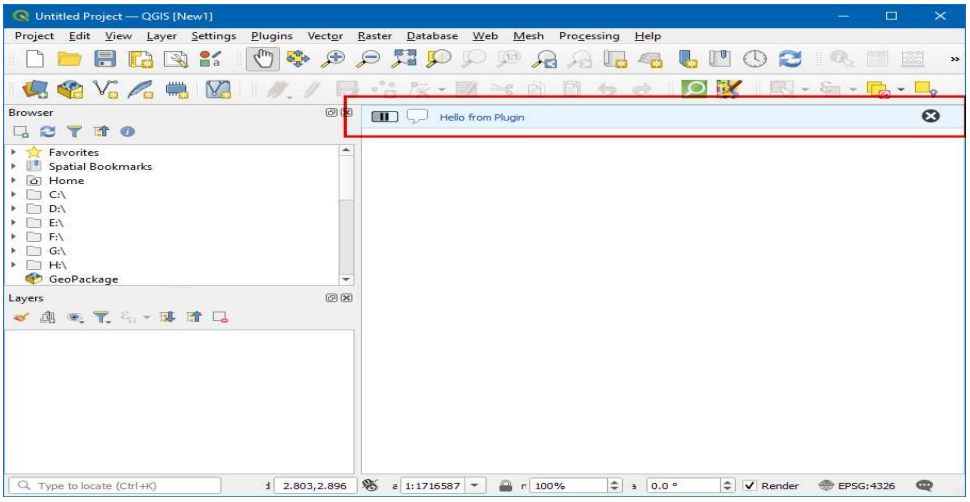
Create these 3 files and put them in a folder named save_attributes. Copy the logo.png file from <home folder>/Downloads/pyqgis_in_a_day/logo.png to this folder. Copy the folder to the python plugins directory at {profile folder}/python/plugins.



Restart QGIS. Go to **Plugins → Manage and Install plugins… → Installed** and enable the **Save Attributes** plugin. You will see the menu entry and the toolbar icon from the plugin.



Now the *Hello from Plugin* message is displayed.



13.2 Processing Plugin

The new and preferred way to write plugins in QGIS is using the Processing Framework. It removes the need for you to design the user interface. The resulting plugin integrates seamlessly in the Processing Toolbox and is inoperable with other processing algorithms.

Let's take the minimal plugin and see what is required to make it a functional processing plugin. We need 2 additional files. One to define the processing algorithm and another to define a new processing provider.

The processing algorithm file is identical to the script we wrote earlier. Create a new file called save_attributes_algorithm.py with the following contents.

save_attributes_algorithm.py

```
from PyQt5.QtCore import QCoreApplication
from qgis.core import (QgsProcessing,
                       QgsProcessingAlgorithm,
                       QgsProcessingParameterFeatureSource,
                       QgsProcessingParameterFileDestination)
```

```
class SaveAttributesAlgorithm(QgsProcessingAlgorithm):
    """Saves the attributes of a vector layer to a CSV file."""
    OUTPUT = 'OUTPUT'
    INPUT = 'INPUT'

    def initAlgorithm(self, config=None):
        self.addParameter(
 QgsProcessingParameterFeatureSource(
                self.INPUT,
                self.tr('Input layer'),
                [QgsProcessing.TypeVectorAnyGeometry]
            )
        )

        # We add a file output of type CSV.
        self.addParameter(
            QgsProcessingParameterFileDestination(
                self.OUTPUT,
                self.tr('Output File'),
                'CSV files (*.csv)',
            )
        )

    def processAlgorithm(self, parameters, context, feedback):
        source = self.parameterAsSource(parameters, self.INPUT, context)
        csv = self.parameterAsFileOutput(parameters, self.OUTPUT, context)

        fieldnames = [field.name() for field in source.fields()]

        # Compute the number of steps to display within the progress bar and
        # get features from source
        total = 100.0 / source.featureCount() if source.featureCount() else 0
        features = source.getFeatures()

        with open(csv, 'w') as output_file:
          # write header
          line = ','.join(name for name in fieldnames) + '\n'
          output_file.write(line)
          for current, f in enumerate(features):
              # Stop the algorithm if cancel button has been clicked
              if feedback.isCanceled():
                  break

              # Add a feature in the sink
              line = ','.join(str(f[name]) for name in fieldnames) + '\n'
              output_file.write(line)

              # Update the progress bar
              feedback.setProgress(int(current * total))

        return {self.OUTPUT: csv}

    def name(self):
        return 'save_attributes'
```

```
    def displayName(self):
        return self.tr('Save Attributes As CSV')

    def group(self):
        return self.tr(self.groupId())

    def groupId(self):
                return ' '

    def tr(self, string):
        return QCoreApplication.translate('Processing', string)

    def createInstance(self):
        return SaveAttributesAlgorithm()
```

Next, we need to define a new processing provider. Create a new file save_attributes_provider.py with the following content.

save_attributes_provider.py

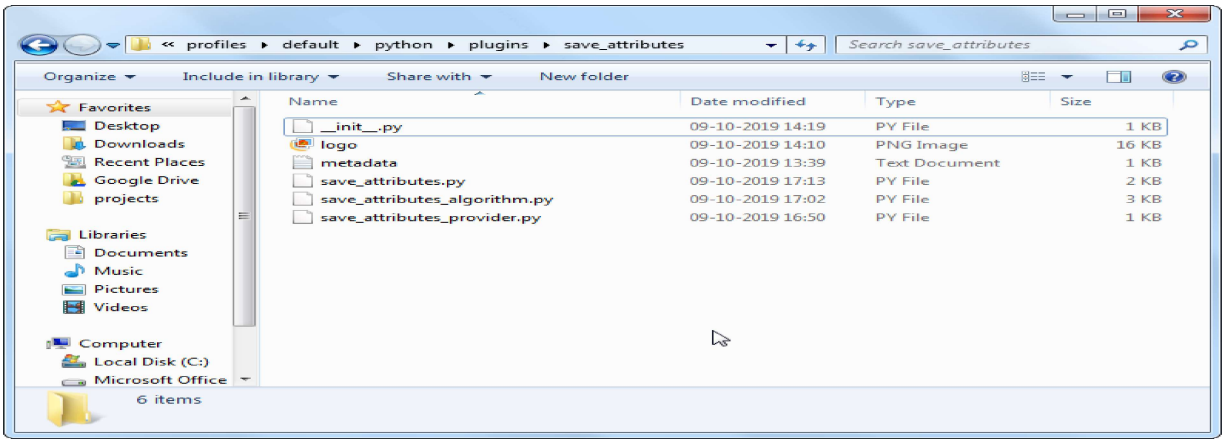
```
import os
import inspect
from PyQt5.QtGui import QIcon

from qgis.core import QgsProcessingProvider
from .save_attributes_algorithm import SaveAttributesAlgorithm


class SaveAttributesProvider(QgsProcessingProvider):

    def __init__(self):
        QgsProcessingProvider.__init__(self)

    def unload(self):
        pass

    def loadAlgorithms(self):
        self.addAlgorithm(SaveAttributesAlgorithm())

    def id(self):
        return 'save_attributes'

    def name(self):
        return self.tr('Save Attributes')

    def icon(self):
        cmd_folder = os.path.split(inspect.getfile(inspect.currentframe()))[0]
        icon = QIcon(os.path.join(os.path.join(cmd_folder, 'logo.png')))
        return icon

    def longName(self):
        return self.name()
```

We now make changes to the existing save_attributes.py file to import and initialize the new processing provider. Change the contents of the file to the following

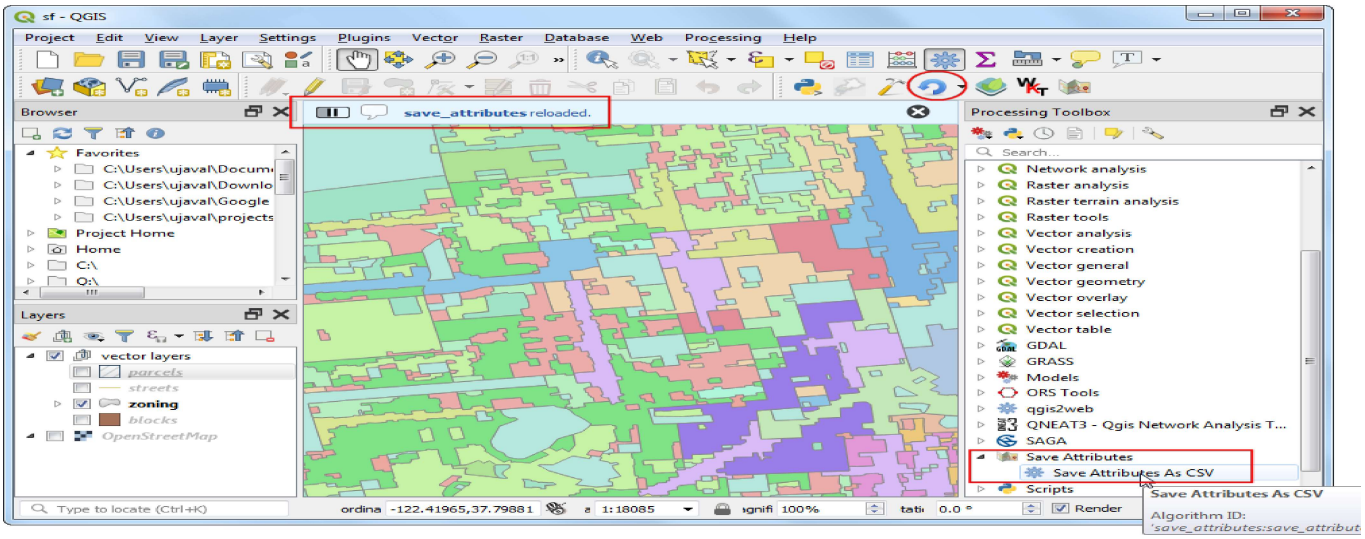
save_attributes.py

```
import os
import sys
import inspect
from PyQt5.QtWidgets import QAction
from PyQt5.QtGui import QIcon

from qgis.core import QgsProcessingAlgorithm, QgsApplication
import processing
from .save_attributes_provider import SaveAttributesProvider


cmd_folder = os.path.split(inspect.getfile(inspect.currentframe()))[0]

class SaveAttributesPlugin:
    def __init__(self, iface):
        self.iface = iface

    def initProcessing(self):
        self.provider = SaveAttributesProvider()
        QgsApplication.processingRegistry().addProvider(self.provider)

    def initGui(self):
        self.initProcessing()
        icon = os.path.join(os.path.join(cmd_folder, 'logo.png'))
        self.action = QAction(QIcon(icon), 'Save Attributes as CSV', self.iface.mainWindow())
        self.action.triggered.connect(self.run)
        self.iface.addPluginToMenu('&Save Attributes', self.action)
        self.iface.addToolBarIcon(self.action)

    def unload(self):
        QgsApplication.processingRegistry().removeProvider(self.provider)
        self.iface.removeToolBarIcon(self.action)
        self.iface.removePluginMenu('&Save Attributes', self.action)
        del self.action

    def run(self):
        processing.execAlgorithmDialog('save_attributes:save_attributes')
```

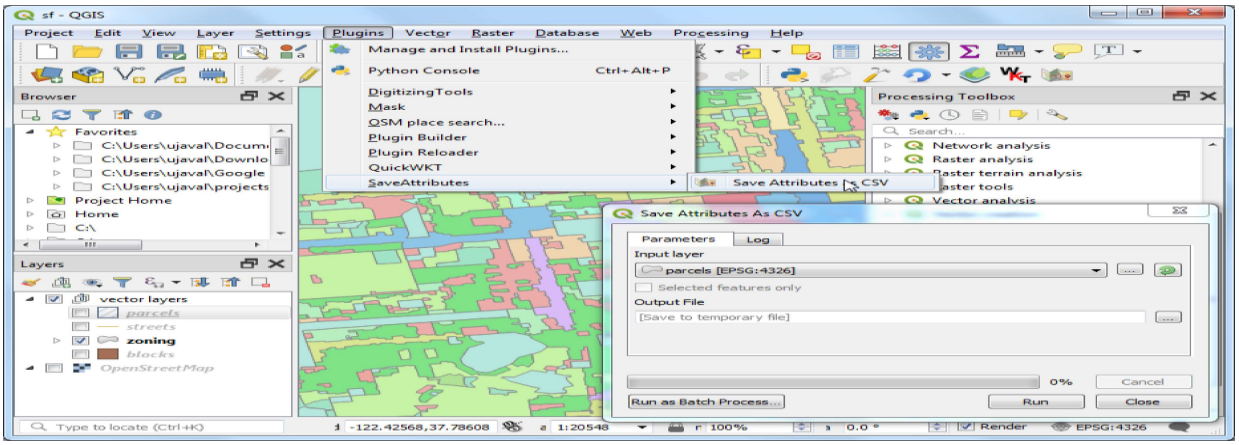
The plugin folder should look like below.



Now let's reload the plugin and see what it looks like. This is a helper plugin which allows iterative development of plugins. Using this plugin, you can change your plugin code and have it reflected in QGIS without having to restart QGIS every time. Find and install the **Plugin Reloader** plugin. Once installed, go to **Plugin → Plugin Reloader → Choose a plugin to be reloaded**.
Select save_attributes in the Configure Plugin reloader dialog.



Once reloaded, you can click on the toolbar button or **Plugin → Save Attributes → Save Attributes As CSV** to launch the processing algorithm.



Writing Standalone Python Scripts

Having the python script run within QGIS is useful and desired most of the time. But there is a way to write python scripts that run on your system without QGIS being open. Ability to run PyQGIS scripts in a headless mode allows you to automate your workflow and run it on a server without human intervention. Let's convert the script from the previous section to a standalone pyqgis script.

Create a new file with the code below and save it as save_attributes_standalone.py.

```
import os
from qgis.core import QgsApplication, QgsVectorLayer, QgsProject
qgs = QgsApplication([], False)
qgs.initQgis()

data_dir = os.path.join(os.path.expanduser('~'), 'Downloads', 'pyqgis_in_a_day')

filename = 'sf.gpkg|layername=zoning'
uri = os.path.join(data_dir, filename)
layer = QgsVectorLayer(uri, 'zoning', 'ogr')

output_name = 'output.csv'
output_path = os.path.join(data_dir, output_name)

with open(output_path, 'w') as output_file:
    fieldnames = [field.name() for field in layer.fields()]
    line = ','.join(name for name in fieldnames) + '\n'
    output_file.write(line)
    for f in layer.getFeatures():
        line = ','.join(str(f[name]) for name in fieldnames) + '\n'
        output_file.write(line)

print('Success: ', 'Output file written at' + output_path)
# Delete the layer object from memory
# Without this you may get a Segmentation Fault on exit
# when the exitQgis() method will try clearning the layer registry
# Alternatively, you can add the layer so it is present in the registry
# QgsProject.instance().addMapLayer(layer, False)
del(layer)
qgs.exitQgis()
```

You will notice that the script is almost exactly the same, but with a few notable changes. First there is the import statement at the top, to explicitly import the required modules. Next, we create an instance of the QgsApplication class and run initQis() method to load the QGIS data providers and layer registry. Finally we call exitQgis() to remove them from memory. Here we don't have a way to take user input, so we hard-code the path to the input layer. Also we don't have the QGIS GUI, we have no way of displaying the messages, so we remove those statements.

When you run the script within QGIS environment, all the paths to QGIS libraries and environment variables are already set and python is able to find and use it. But when you run the script outside of QGIS, you need to set them yourself. The way to run the script depends on the platform.

You may need to change these paths slightly based on where QGIS is installed. The following scripts assume you are running QGIS 3.16 LTR installed at the default location.

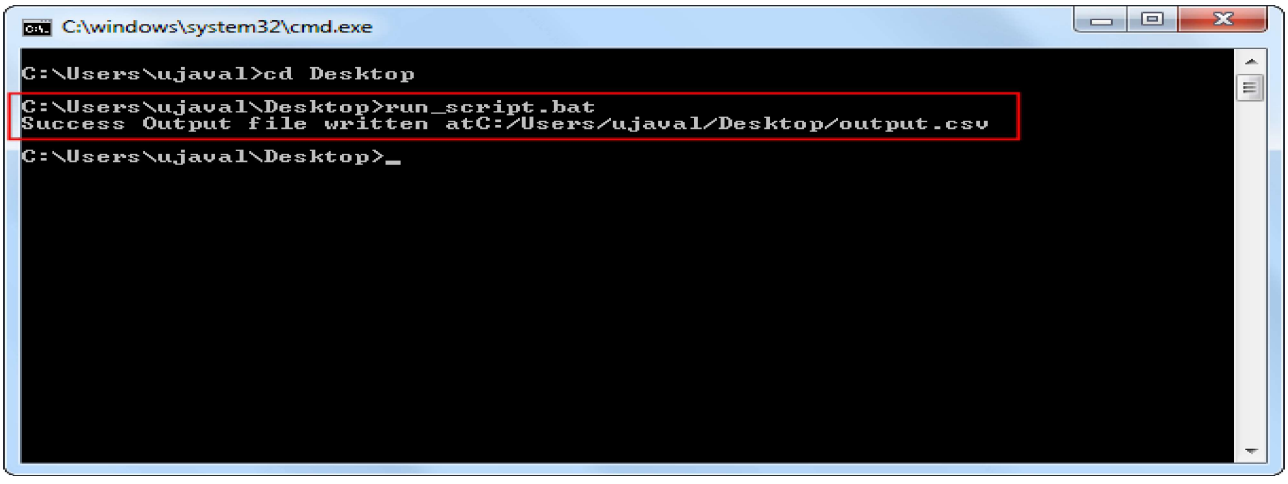
Windows Configuration

On Windows, you can do this using a batch file. Create a new file named run_script.bat with the following code. Make sure to save it in the same directory.

```
@echo off

set OSGEO4W_ROOT=C:\OSGeo4W64

call "%OSGEO4W_ROOT%\bin\o4w_env.bat"
call "%OSGEO4W_ROOT%\bin\qt5_env.bat"
call "%OSGEO4W_ROOT%\bin\py3_env.bat"

set PATH=%OSGEO4W_ROOT%\bin;%OSGEO4W_ROOT%\apps\qgis-ltr\bin;C:\OSGeo4W64\apps\Qt5\bin;%PATH%
set PYTHONPATH=%OSGEO4W_ROOT%\apps\qgis-ltr\python;%OSGEO4W_ROOT%\apps\qgis-ltr\python\plugins;%PYTHONPATH%
set QGIS_PREFIX_PATH=%OSGEO4W_ROOT%\apps\qgis-ltr
set QT_QPA_PLATFORM_PLUGIN_PATH=%OSGEO4W_ROOT%\apps\Qt5\plugins

python3 save_attributes_standalone.py
```

Open the command prompt, browse to the directory with the above files, type run_script.bat and press Enter.

You can also just double-click the run_script.bat to run it, but you will not see any error or success messages. So it is always a good idea to run the script from the shell.
The script will run and produce the output at the given path.



15. Supplement

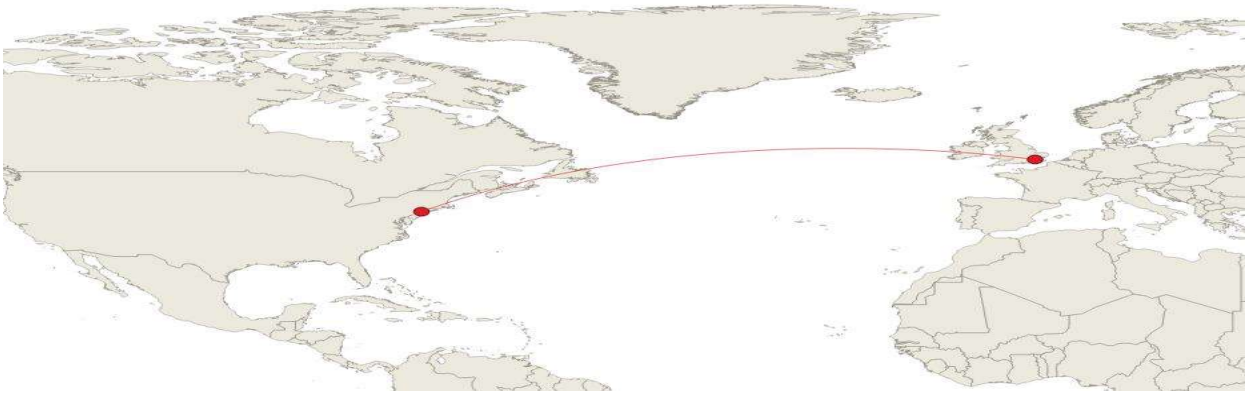
This section contains code snippets for common PyQGIS operations.

All code snipets assume you have loaded the sf.qgz project. You can load the project using the code below.

```
import os
data_dir = os.path.join(os.path.expanduser('~'), 'Downloads', 'pyqgis_in_a_day')
project = QgsProject.instance()
project_name = 'sf.qgz'
project_path = os.path.join(data_dir, project_name)
project.read(project_path)
```

Create a Geodesic Line

Connects 2 points using the great circle arc along an ellipsoid. Uses the geodesicLine() method provided by the QgsDistanceArea() class for the computation.



```
from qgis.core import QgsDistanceArea

new_york = (40.661, -73.944)
london = (51.5072, 0.1276)

lat1, lon1 = new_york
lat2, lon2 = london

# Remember the order is X,Y
point1 = QgsPointXY(lon1, lat1)
point2 = QgsPointXY(lon2, lat2)

d = QgsDistanceArea()
d.setEllipsoid('WGS84')

# Create a geodesic line with vertices every 100km
vertices = d.geodesicLine(point1, point2, 100000)

# Create a polyline from the vertices
# The method returns 1 geometry unless the line crosses antimeridien
geodesic_line = QgsGeometry.fromPolylineXY(vertices[0])

# Create a line layer to display the route
vlayer = QgsVectorLayer('LineString?crs=EPSG:4326', 'route', 'memory')
provider = vlayer.dataProvider()

f = QgsFeature()
f.setGeometry(geodesic_line)
provider.addFeature(f)
```

```
vlayer.updateExtents()
QgsProject.instance().addMapLayer(vlayer)
```

Save Map Rendering as an Image

saveAsImage() method is quick and easy, but you do not have much control over the resulting image. You can't control the resolution, size or how each layer will be rendered. There is another way to achieve this. You can look at the code for exporting map as an image in QGIS and you will discover 2 classes QgsMapRendererParallelJob and QgsMapRendererSequentialJob that lets you achieve a better result. The code snippet below exports a hi-resolution image of the project. [4]

```
import os
data_dir = os.path.join(os.path.expanduser('~'), 'Downloads', 'pyqgis_in_a_day')
image_name = 'sf_hires.png'
image_path = os.path.join(data_dir, image_name)

settings = iface.mapCanvas().mapSettings()
settings.setOutputSize(QSize(1000,1000))

settings.setFlag(QgsMapSettings.DrawLabeling, False)
settings.setFlag(QgsMapSettings.Antialiasing, True)

job = QgsMapRendererSequentialJob(settings)
job.start()
job.waitForFinished()
image = job.renderedImage()
image.save(image_path)
```

Turn a layer on/off

```
blocks = QgsProject.instance().mapLayersByName('blocks')[0]
QgsProject.instance().layerTreeRoot().findLayer(blocks.id()).setItemVisibilityChecked(True)
```

Get all Layers

```
for layer in QgsProject.instance().mapLayers().values():
```

```
    print(layer.name())
```

Get only checked (visible) Layers

```
for layer in iface.mapCanvas().layers():
    print(layer.name())
```

Get only selected Layers

```
for layer in iface.layerTreeView().selectedLayers():
  print(layer.name())
```

Set Canvas Extent to a Layer Extent

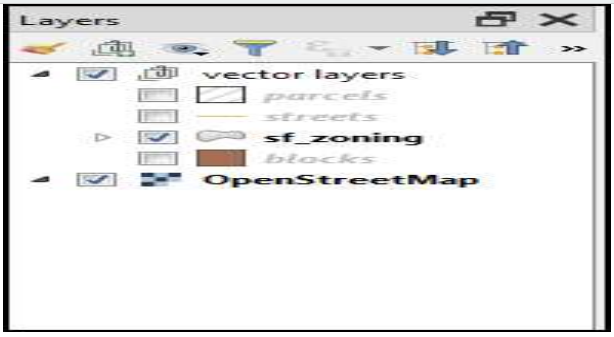
Select one of the layers in the *Layers* panel and run the following code.

```
layer = iface.activeLayer()
mc = iface.mapCanvas()
mc.setExtent(layer.extent())
mc.refresh()
```

Get Layers with Hierarchy

This code snippet is taken from the Cheat Sheet for PyQGIS , but contains an important modification. If you notice carefully, the function getGroupLayers is called recursively from within getGroupLayers. This allows one to even get layers that have sub-groups within layer groups.

```
def getGroupLayers(group):
    print('- group:' + group.name())
    for child in group.children():
        if isinstance(child, QgsLayerTreeGroup):
            getGroupLayers(child)
        else:
            print('  - layer:' + child.name())

root = QgsProject.instance().layerTreeRoot()
for child in root.children():
    if isinstance(child, QgsLayerTreeGroup):
        getGroupLayers(child)
    elif isinstance(child, QgsLayerTreeLayer):
        print ("- layer: " + child.name())
```



Remove a Specific Button from a Toolbar

The following code snippet shows how once can find names of different widget in the QGIS Application and locate specific toolbars and button. The following code will disable to *Deselect All Features from Layers* button from the *Selection Toolbar*.

```
# Find all child objects of the mainWindow that are type QToolbar
for x in iface.mainWindow().findChildren(QToolBar):
    print(x.objectName())

# In the printed list, we observe that
# the selection toolbar is named mSelectionToolBar
# Get a reference to it by name
selectionToolbar = iface.mainWindow().findChild(QToolBar,'mSelectionToolBar')

# Find the buttons on the toolbar
for x in selectionToolbar.findChildren(QAction):
    print(x.objectName())
# Alternatively, you can also use the following
for x in selectionToolbar.actions():
    print(x.objectName())

# If the toolbar button is not an action, you
# We know the name of the deselection button
# Get a reference to it by name
deselection = selectionToolbar.findChild(QAction,'ActionDeselection')

# We can disable a widget by calling setEnabled(False)
deselection.setEnabled(False)

# We can remove it completely by calling removeAction()
selectionToolbar.removeAction(deselection)
```

Add a Drop-down Menu to Toolbar

This example shows how to add a QCombobox widget to a toolbar and populate it with attribute values from a layer. On click of the button, the selection is read from the combo box and used to apply a filter to the layer

You must have a layer with an attribute called **NAME** for this example to work. Tested with the Natural Earth Admin 0 - Countries layer.

```
toolBar = iface.addToolBar("My Toolbar")
toolBar.setObjectName("My Toolbar")

countryCombo = QComboBox(toolBar)
toolBar.addWidget(countryCombo)
countryCombo.setToolTip("Select a country")


filterButton = QPushButton('Filter')
filterButton.setToolTip('Filter')
toolBar.addWidget(filterButton)

resetButton = QPushButton('Reset')
```

```
resetButton.setToolTip('Reset')
toolBar.addWidget(resetButton)

# We want to sort the layer by country names
# configure QgsFeatureRequest and use it with getFeatures()
request = QgsFeatureRequest()
clause = QgsFeatureRequest.OrderByClause('NAME')
orderby = QgsFeatureRequest.OrderBy([clause])
request.setOrderBy(orderby)
layer = iface.activeLayer()

# Read the country names and save in a list
countries = []
for f in layer.getFeatures(request):
  countries.append(f['NAME'])
# Add items to the combobox
for country in countries:
    countryCombo.addItem(country)

# Define functions to apply and reset filters
def filter_layer():
    layer = iface.activeLayer()
    country = countryCombo.currentText()
    expression = '\"NAME\" = \'{}\'".format(country)
    layer.setSubsetString(expression)

def reset_layer():
    layer = iface.activeLayer()
    layer.setSubsetString('')
    countryCombo.clear()
    for country in countries:
      countryCombo.addItem(country)

filterButton.clicked.connect(filter_layer)
resetButton.clicked.connect(reset_layer)
```

Adding CSV Layers

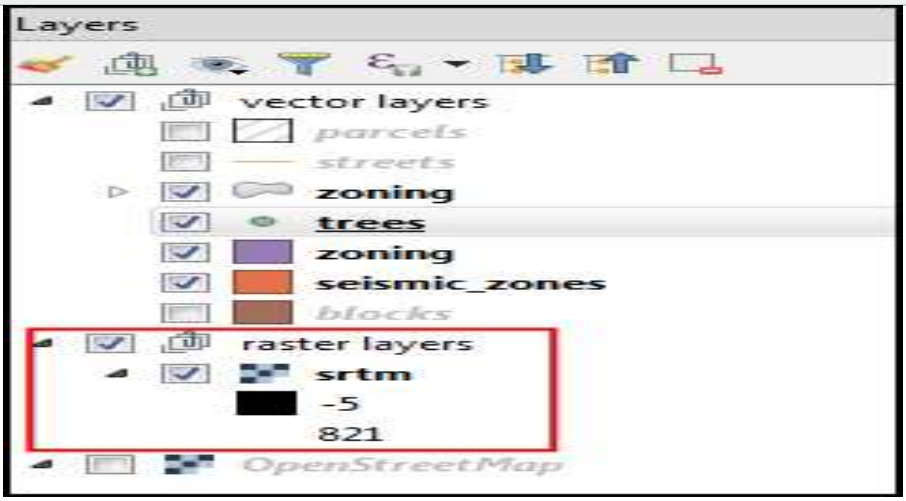
```
import os
data_dir = os.path.join(os.path.expanduser('~'), 'Downloads', 'pyqgis_in_a_day')

filename = 'trees.csv'
csvpath = 'file:///' + data_dir + filename
uri = '{}?type=csv&xField={}&yField={}&crs={}'.format(
  csvpath, 'Longitude', 'Latitude', 'EPSG:4326')
iface.addVectorLayer(uri, 'trees', 'delimitedtext')
```

Inserting Layers in the Layer Tree

We can use the QgsLayerTree class to insert the layer at an appropriate place.

```
import os
data_dir = os.path.join(os.path.expanduser('~'), 'Downloads', 'pyqgis_in_a_day')
```

```
filename = 'srtm.tif'
uri = os.path.join(data_dir, filename)
rlayer = QgsRasterLayer(uri, 'srtm', 'gdal')

rastergroup = QgsLayerTreeGroup('raster layers')
treelayer = QgsLayerTreeLayer(rlayer)
rastergroup.insertChildNode(0, treelayer)

root = QgsProject.instance().layerTreeRoot()
root.insertChildNode(1, rastergroup)
```



Create a New Vector Layer

A simple example showing how to create a point layer with 1 feature and 1 attribute. [5]

```
vlayer = QgsVectorLayer('Point?crs=EPSG:4326', 'point', 'memory')

provider = vlayer.dataProvider()
provider.addAttributes([QgsField('name', QVariant.String)])
vlayer.updateFields()
f = QgsFeature()
f.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(-122.41, 37.77)))
f.setAttributes(['San Francisco'])
provider.addFeature(f)
vlayer.updateExtents()
QgsProject.instance().addMapLayer(vlayer)
```

Saving Layers to Disk

Use the QgsRasterFileWriter or QgsVectorFileWriter classes for writing layers to disk. [6]

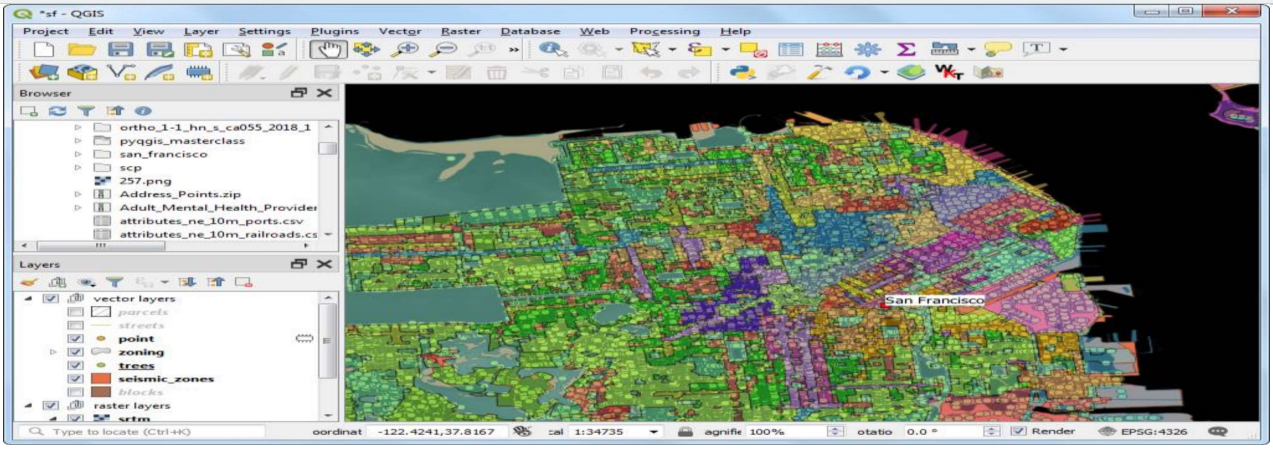
```
import os
data_dir = os.path.join(os.path.expanduser('~'), 'Downloads', 'pyqgis_in_a_day')
options = QgsVectorFileWriter.SaveVectorOptions()
options.actionOnExistingFile = QgsVectorFileWriter.CreateOrOverwriteLayer
options.layerName = 'point'
filename = 'sf.gpkg'
path = os.path.join(data_dir, filename)
```

```
QgsVectorFileWriter.writeAsVectorFormat(vlayer, path, options)
```

Displaying a label with a background color

This is a vast topic, but you can get a taste of the flexibility offered by the API to control all aspects of labeling. Notice the class name QgsPalLayerSettings - this is because QGIS uses a labeling library called PAL for labels. The code snippet below shows how to create a label for a point layer with a background color. [7]

```
vlayer = QgsProject.instance().mapLayersByName('point')[0]
symbol = QgsMarkerSymbol.createSimple({'name': 'square', 'color': 'red'})
vlayer.renderer().setSymbol(symbol)

label_settings = QgsPalLayerSettings()
#label_settings.drawBackground = True
label_settings.fieldName = 'name'

text_format = QgsTextFormat()
background_color = QgsTextBackgroundSettings()
background_color.setFillColor(QColor('white'))
background_color.setEnabled(True)
text_format.setBackground(background_color )
label_settings.setFormat(text_format)

vlayer.setLabeling(QgsVectorLayerSimpleLabeling(label_settings))
vlayer.setLabelsEnabled(True)
vlayer.triggerRepaint()
```



Edit Attribute Table of a Vector Layer

When you use processing, a new layer is created by each algorithm. This example shows, how to use processing to overwrite the original layer with the results of processing.

```
import os
data_dir = os.path.join(os.path.expanduser('~'), 'Downloads', 'pyqgis_in_a_day')
filename = 'sf.gpkg|layername=blocks'
uri = os.path.join(data_dir, filename)
blocks = QgsVectorLayer(uri, 'blocks', 'ogr')
```