PRACTICAL LECTURE
Plug-in development in QGIS

Fredrik Lindberg

Urban Climate Group
Department of Earth Sciences
University of Gothenburg

Plug-ins in QGIS

Plug-ins is one possibility for the user-community to add to value to the QGIS-project. Some plugins are incorporated into the core software.

It is possible to create plugins in Python programming language. In comparison with classical plugins written in C++ these should be easier to write, understand, maintain and distribute due the dynamic nature of the Python language.

There are two (three) options:

1. Write a python plugin – goes into the plugin menu
2. Write a C++ plugin – (I don't know about this one)
3. Write a processing plugin – goes into the processing toolbox

Plug-in structures

- User plug-ins are located in **C:\Users\YOURUSERNAME\AppData\Roaming\QGIS\QGIS3\profiles\default\python\plugins**

FILES NEEDED:

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

Pre-preparations

Prepare a code repository to share your code and to users to report issues. We will make us of GitHub (www.github.com)

- If you do not have an account, create one
- Create a new repository called **UAVPreparer**

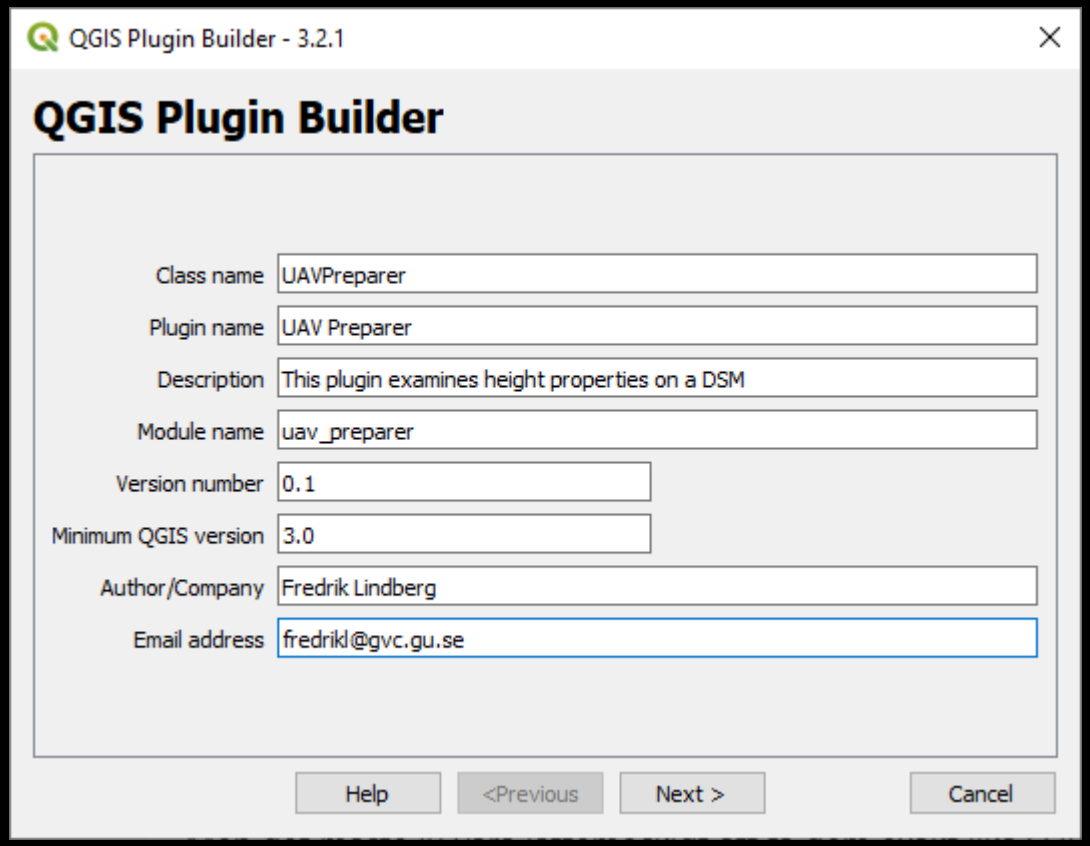
We will make use of other plug-ins that can be used for plug-in development

- Install **Plugin Builder** from the QGIS plug-in repository
- Install **Plugin Reloader** from the QGIS plug-in repository

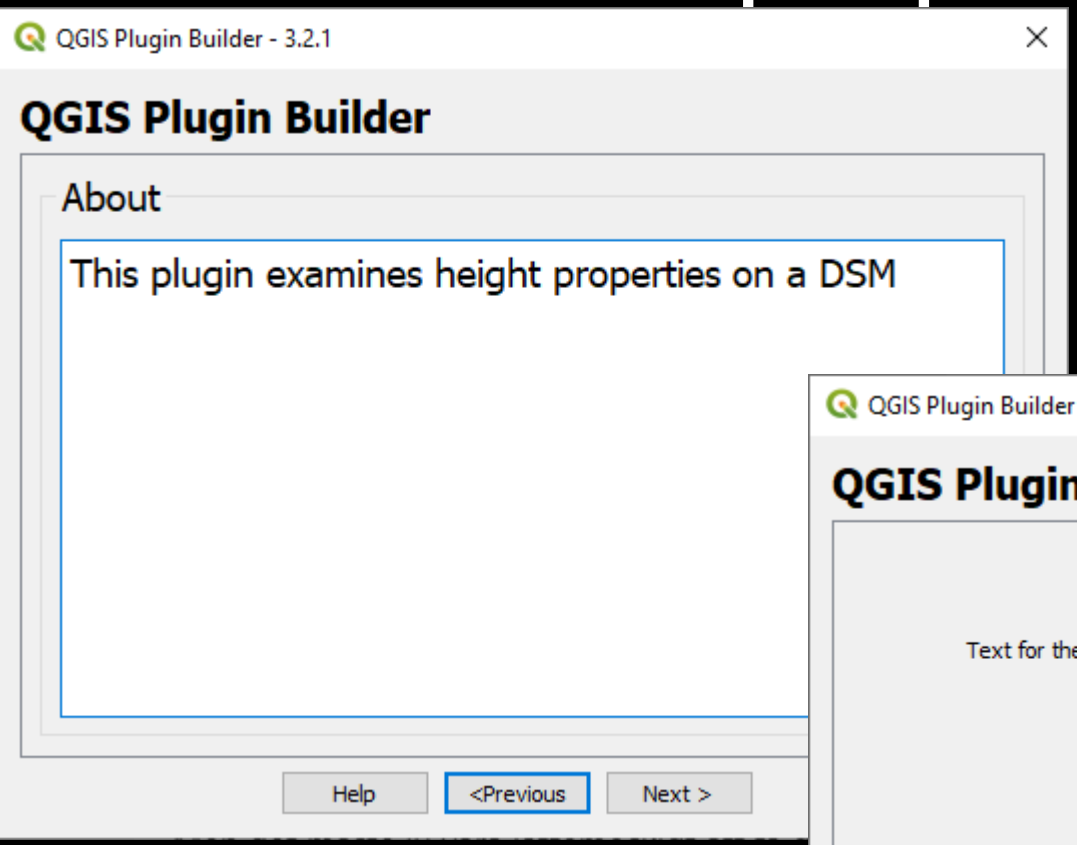
We will also make use of **pb_tool** for deploying etc. of our plug-in

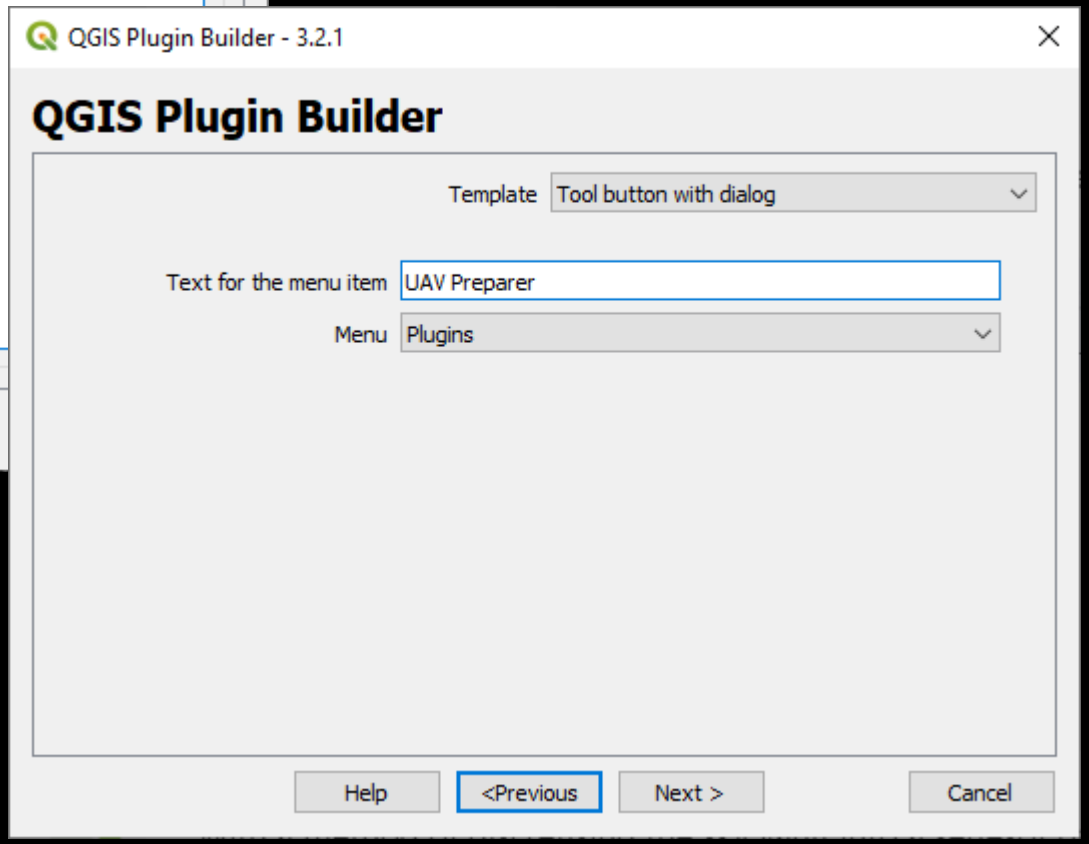
Pre-preparations

- Run the Plugin Builder and make the following setting:

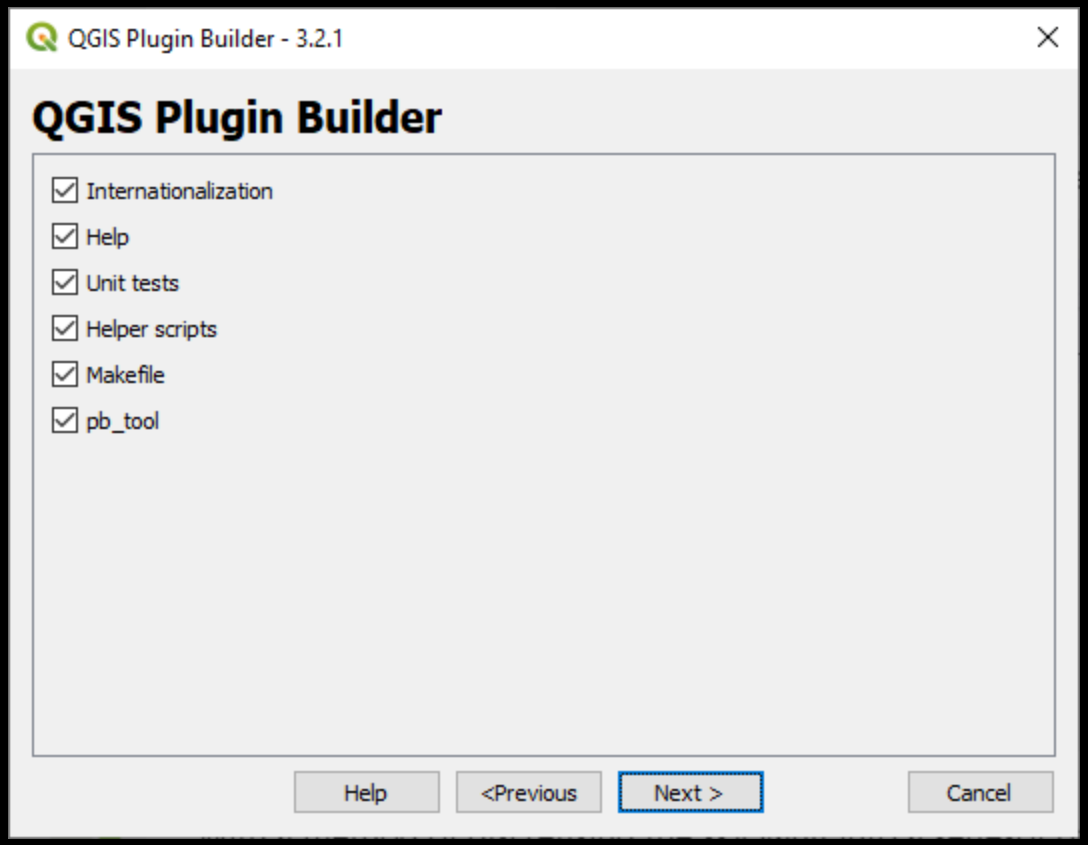


Pre-preparations

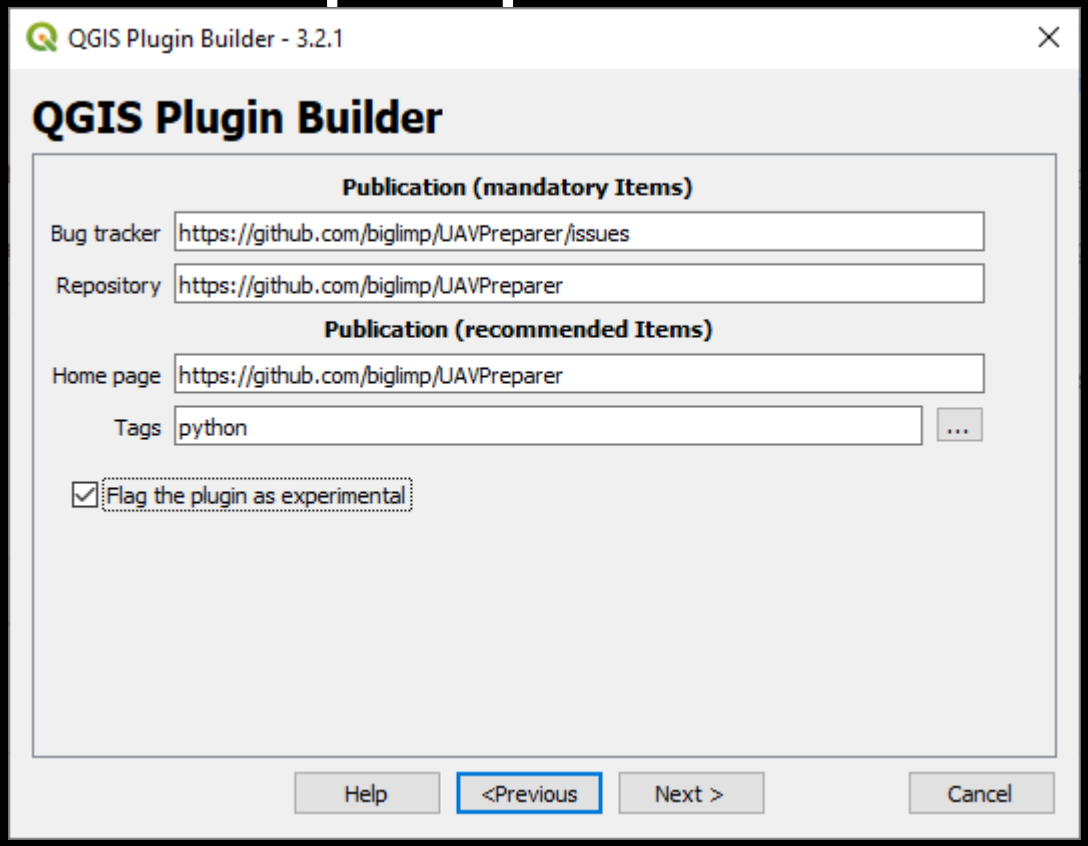




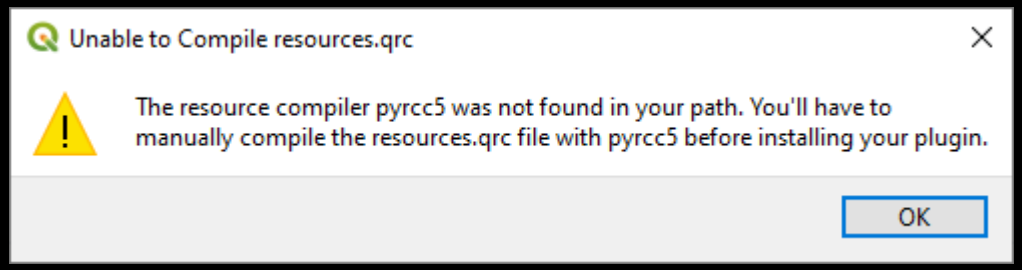
Pre-preparations



Pre-preparations

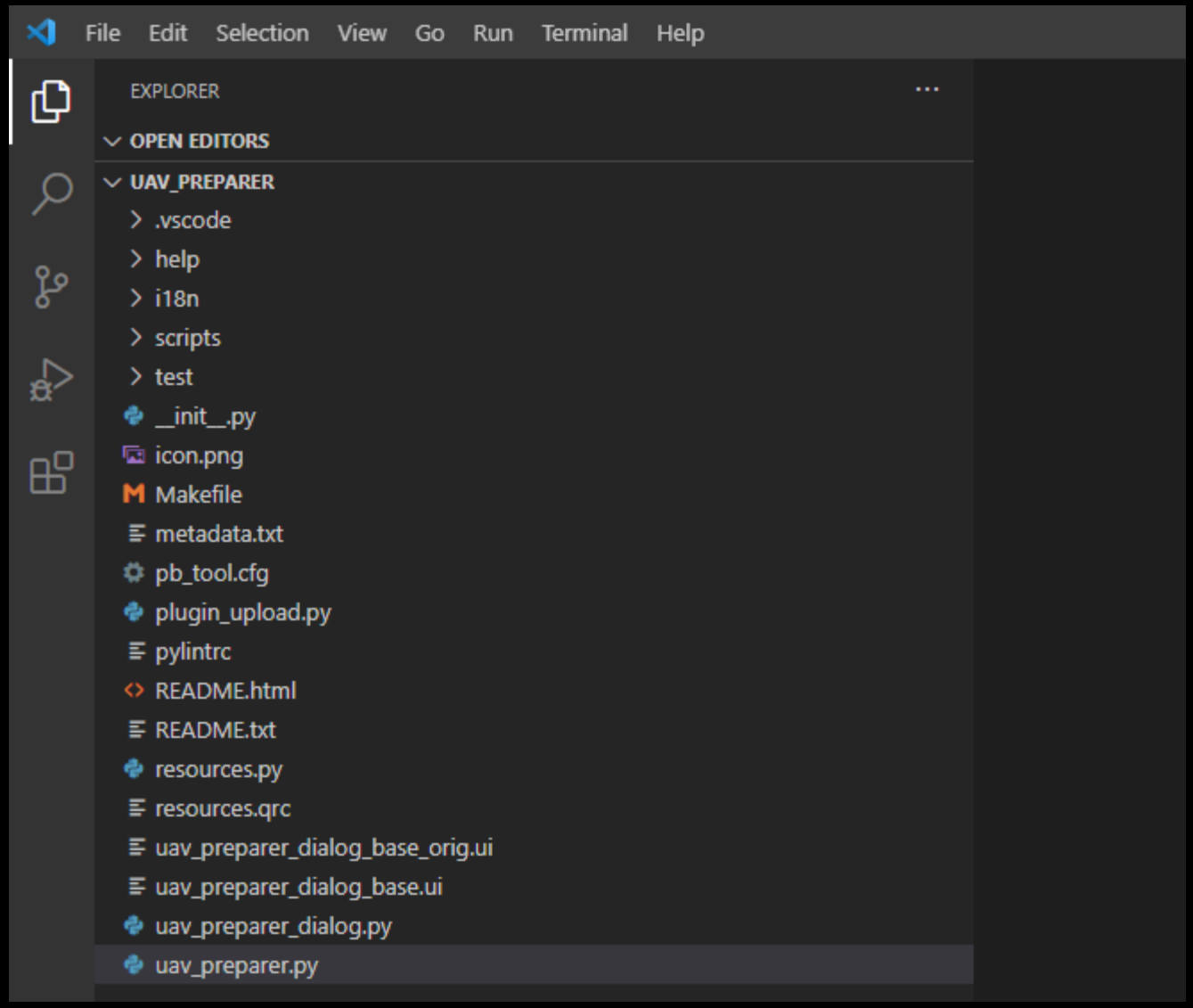


Save on a location with read and write on your computer. Ignore message. This will be fixed soon.



VSCode

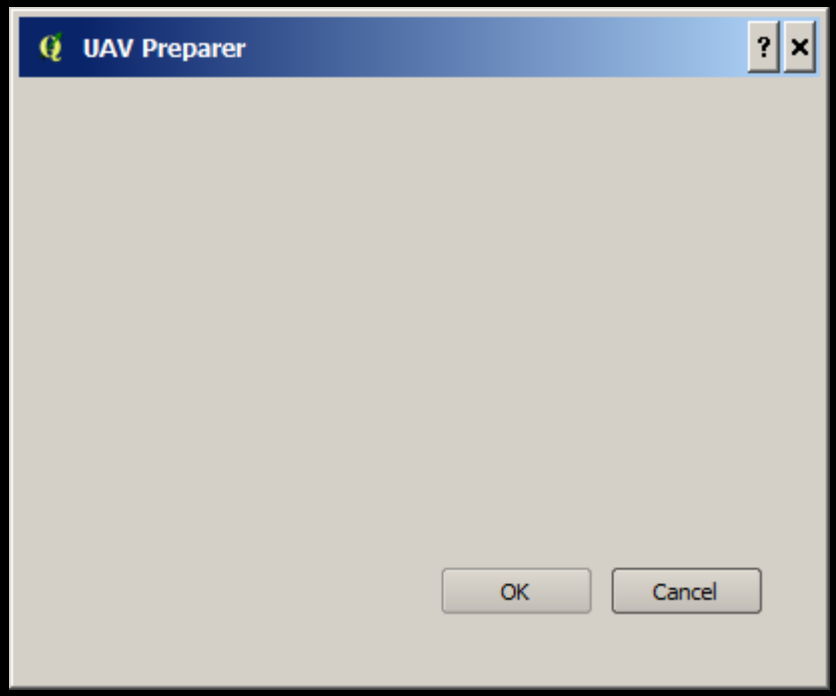
Start VSCode with your .bat-file and open the folder where you saved your plugin



VSCode

First, see is **pb_tool** is installed correctly.

- Start the terminal in VSCode and type type **pb_tool**
- If successful, type **pb_tool deploy**. This moves your plug-in to the plug-in folder
- Restart QGIS and activate your plug-in

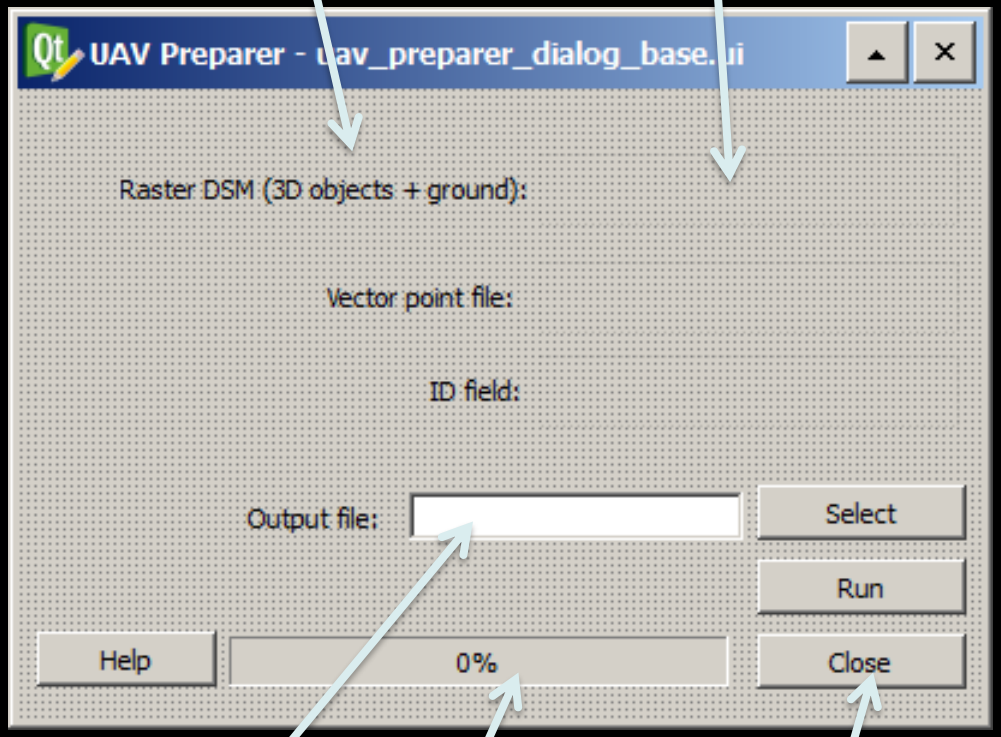


The ui-file

Now, adjust our ui-file by double-click on **uav_preparer_dialog_base.ui**. This will open **QtDesigner**. Make it look like the figure below:

Label

Widget (an object where other object can be located)

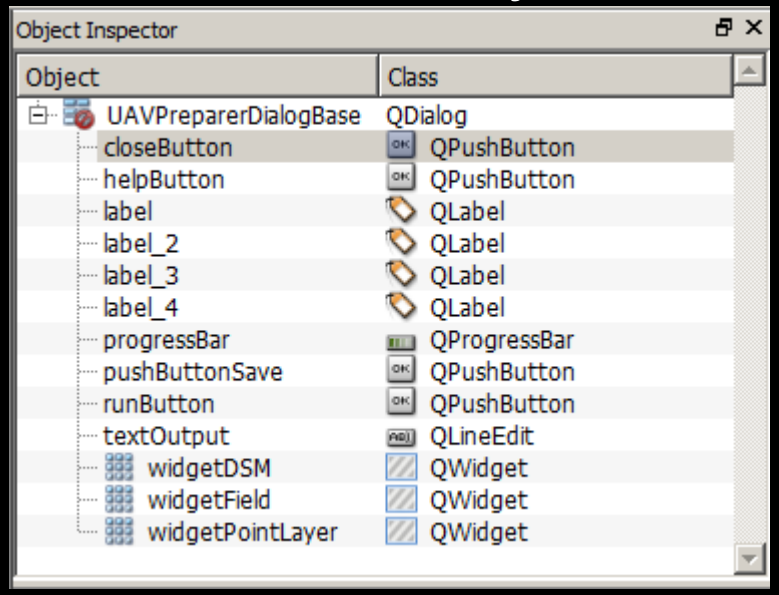


Line edit

Progress bar

Push button

Name of the objects:



Start adding code

- Open **uav_preparer.py** and add the following code before self.dlg.show() in the run function to access the raster layer:

```
self.layerComboManagerDSM = QgsMapLayerComboBox(self.dlg.widgetDSM)
self.layerComboManagerDSM.setFilters(QgsMapLayerProxyModel.RasterLayer)
self.layerComboManagerDSM.setFixedWidth(175)
self.layerComboManagerDSM.setCurrentIndex(-1)
```

- To access the vector layer and the field:

```
self.layerComboManagerPoint = QgsMapLayerComboBox(self.dlg.widgetPointLayer)
self.layerComboManagerPoint.setCurrentIndex(-1)
self.layerComboManagerPoint.setFilters(QgsMapLayerProxyModel.PointLayer)
self.layerComboManagerPoint.setFixedWidth(175)
self.layerComboManagerPointField = QgsFieldComboBox(self.dlg.widgetField)
self.layerComboManagerPointField.setFilters(QgsFieldProxyModel.Numeric)
self.layerComboManagerPoint.layerChanged.connect(self.layerComboManagerPoint
Field.setLayer)
```

More code

- Add a function to save file:

```
def savefile(self):
    self.outputfile = self.fileDialog.getSaveFileName(None, "Save File As:", None, "Text Files (*.txt)")
    self.dlg.textOutput.setText(self.outputfile[0])
```

- Also add in the run function:

```
# Set up of file save dialog
    self.fileDialog = QFileDialog()
    self.dlg.pushButtonSave.clicked.connect(self.savefile)
```

- Make sure that you import the correct libraries

And more code

- Add a function to help button:

```
def help(self):
        url = "https://github.com/biglimp/UAVPreparer"
        webbrowser.open_new_tab(url)
```

Also add in the run function:

```
# Set up of file save dialog
    self.fileDialog = QFileDialog()
    self.dlg.pushButtonSave.clicked.connect(self.savefile)
```

Add start_progress function and connect in the run function:

```
# Set up for the Run button
self.dlg.runButton.clicked.connect(self.start_progress)
```

Add start_progress function and connect in the run function:

Put your main code in the start_progress function

Task 1

Complete the plug-in by adding (and adjusting) your code from previous tutorials

- To control the progress bar:

```
self.dlg.progressBar.setRange(0, numfeat)
self.dlg.progressBar.setValue(i + 1)
```

- Use ***i*** as an index in the for loop since ***f*** is not a number but a Qfeature
- Use a Signal/Slot in QtDesigner so that the plugin close when clicking on **Close** button
- Communicate with user if something goes wrong when adding layers:

```
point_layer = self.layerComboManagerPoint.currentLayer()
if point_layer is None:
    QMessageBox.critical(None, "Error", "No valid vector point layer is selected")
    return
else:
    vlayer = QgsVectorLayer(point_layer.source(), "polygon", "ogr")
```

Task 2

Add the possibility to change the search radius around each point

Task 3

Upload your code to your repository

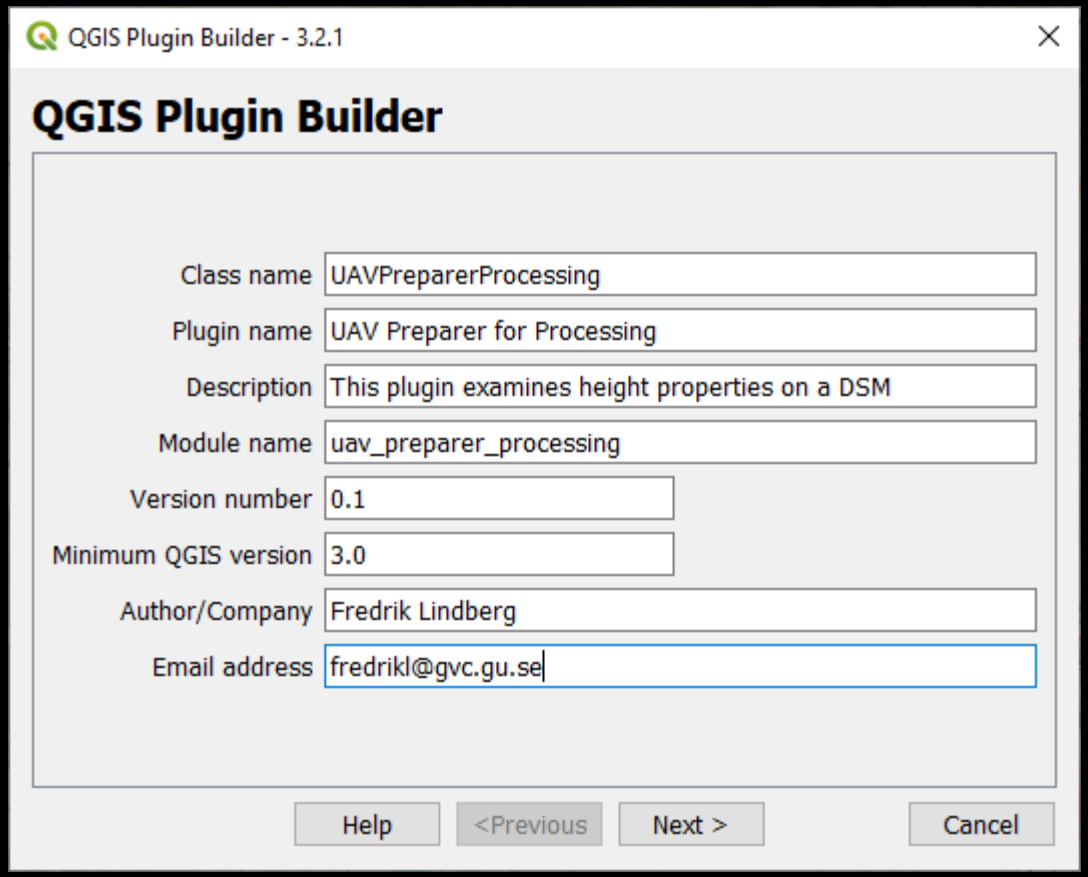
Making a processing plugin

Advantages of making a plugin using the processing framework:

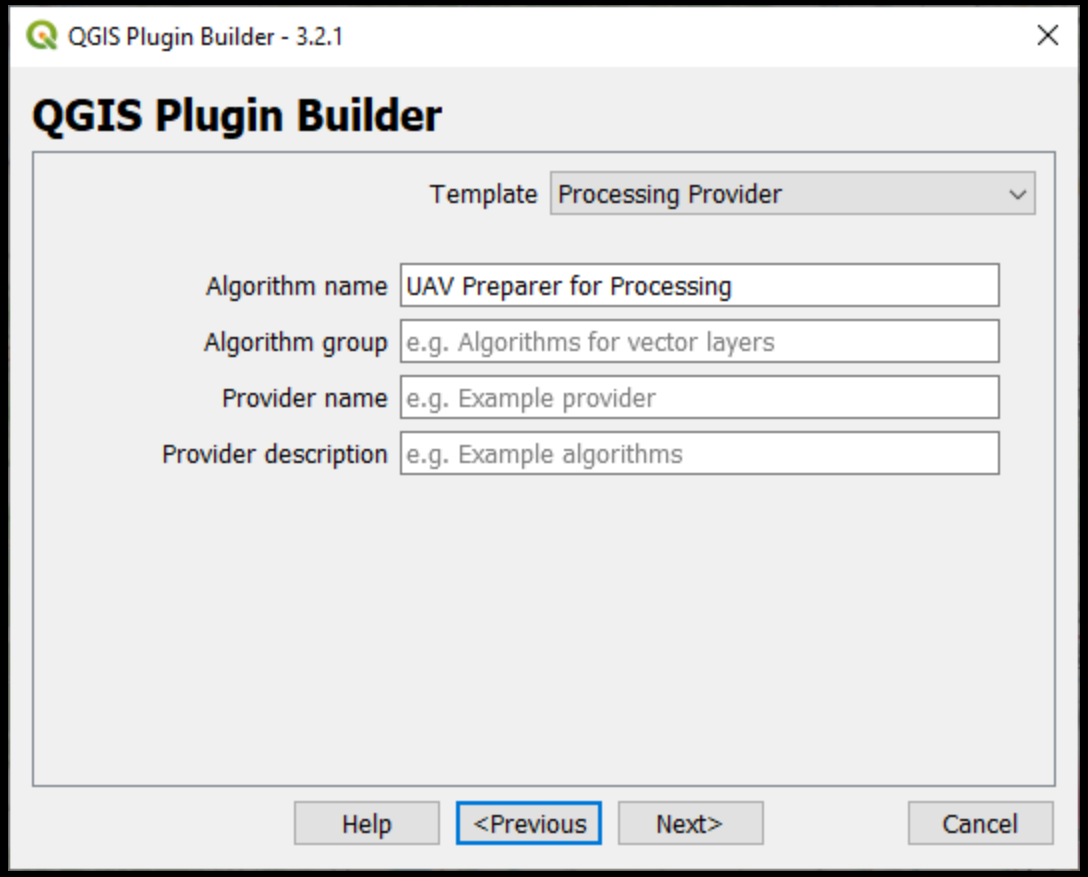
- A common API makes the plugin more accessible, e.g. for the **model builder**, as a **batch process** or as a **standalone Python function**
- Your plugin automatically gets ported to a thread on your computer (computer will not freeze)
- No ui-file required (could also be disadvantage)

Pre-preparations

- Run the Plugin Builder and make the same settings as above with the following changes:



Pre-preparations



Start adding code

In uav_preparer_processing_provider.py:

In id function: return ‘uavpreparer’

In name function: return 'UAV Prepearer‘

In pb_tool.cfg:

python_files: __init__.py uav_preparer_processing.py uav_preparer_processing_provider.py uav_preparer_processing_algorithm.py

Run pb_tool deploy in the terminal and activate the plugin in QGIS.

DON’T FORGET TO ADD IMPORTS IF NEEDED!

Adding parameters to the plugin

In uav_preparer_processing_algorithm.py:

Replace OUTPUT AND INPUT line 59-60 with:

```
INPUT_DSM = 'INPUT_DSM'
INPUT_POINT = 'INPUT_POINT'
ID_FIELD = 'ID_FIELD'
RADIUS = 'RADIUS'
OUTPUT_FILE = 'OUTPUT_FILE‘
```

Adding parameters to the plugin

In initAlgorithm function (what the user will see):

```
self.plugin_dir = os.path.dirname(__file__) #needed later to save temporary files

self.addParameter(
    QgsProcessingParameterRasterLayer(
        self.INPUT_DSM,
        self.tr('Digital Surface Model'),
    )
)

self.addParameter(
    QgsProcessingParameterFeatureSource(
        self.INPUT_POINT,
        self.tr('Input Point layer'),
        [QgsProcessing.TypeVectorPoint]
    )
)
```

Adding parameters to the plugin

In initAlgorithm function (cont.):

```
self.addParameter(
    QgsProcessingParameterField(
        self.ID_FIELD,
        self.tr('ID field'),
        '',
        self.INPUT_POINT,
        QgsProcessingParameterField.Numeric
    )
)

self.addParameter(
    QgsProcessingParameterNumber(
        self.RADIUS,
        self.tr('Radius (m)'),
        QgsProcessingParameterNumber.Double,
        100, False))
```

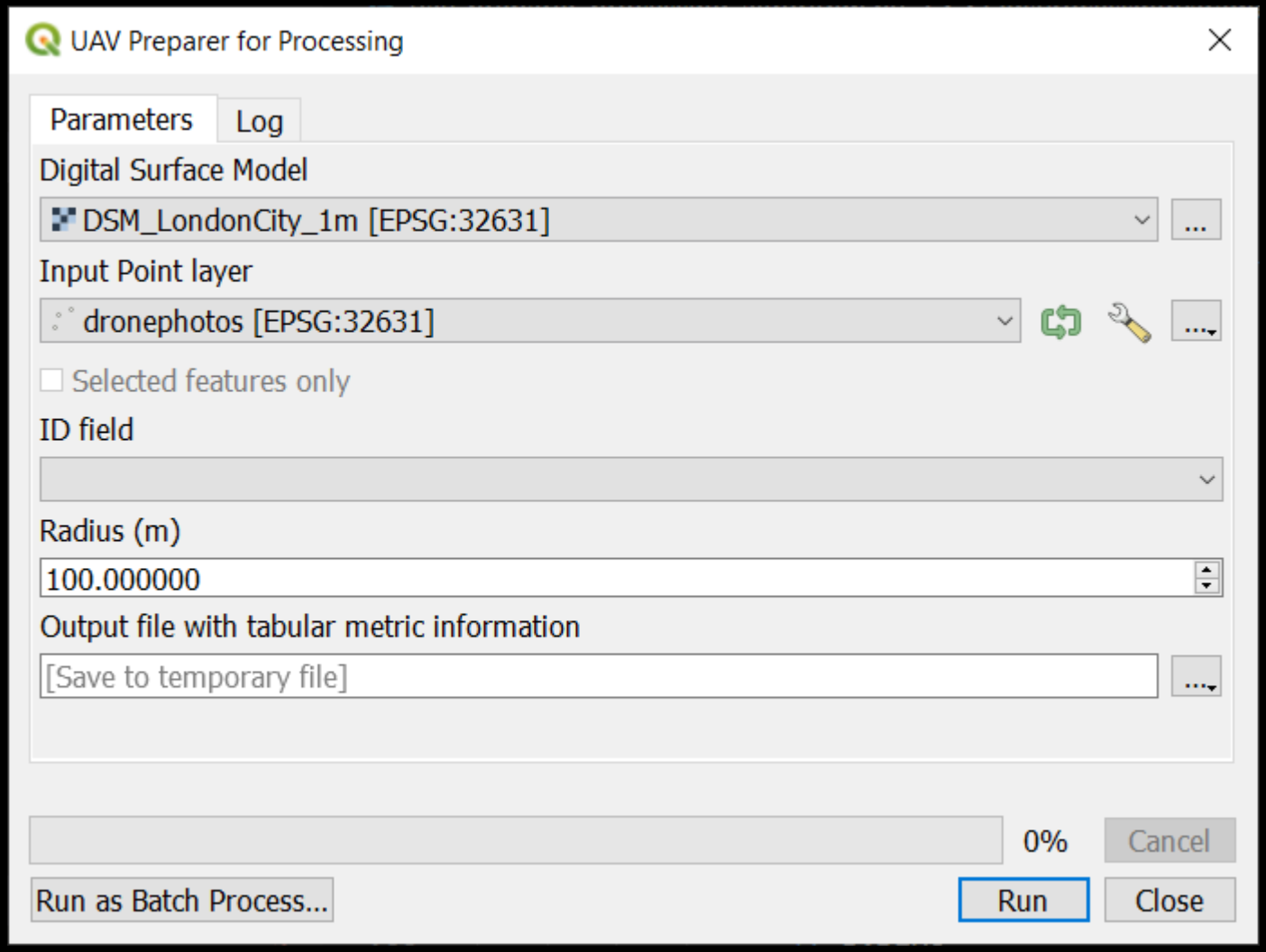
Adding parameters to the plugin

In initAlgorithm function (cont.):

```
self.addParameter(
    QgsProcessingParameterNumber(
        self.RADIUS,
        self.tr('Radius (m)'),
        QgsProcessingParameterNumber.Double,
        100,
        False
    )
)
self.addParameter(
    QgsProcessingParameterFileDestination(
        self.OUTPUT_FILE,
        self.tr('Output file with tabular metric information'),
        self.tr('TXT files (*.txt *.txt)')))
```

Adding parameters to the plugin

Your interface should something like:



Adding parameters to the plugin

In processAlgorithm function (what the plugin will do). **Plugin_builder** have already added a loop. See how feedback is given back to user. feedback.setProgressText() can also be used to communicate.

First we need to access our input (DSM, Pointfile and ID, Radius and output):

```
# InputParameters
vlayer = self.parameterAsVectorLayer(parameters, self.INPUT_POINT, context)
idField = self.parameterAsFields(parameters, self.ID_FIELD, context)
dsmlayer = self.parameterAsRasterLayer(parameters, self.INPUT_DSM, context)
radius = self.parameterAsDouble(parameters, self.RADIUS, context)
outputFile = self.parameterAsFileOutput(parameters, self.OUTPUT_FILE, context)
```

Task

Now try to add you code from the earlier plugin to finalize your processing tool.

Output command should look like: `return {self.OUTPUT_FILE: outputFile}`

Task2

Upload code to a new branch in your repository