
Create QGIS Plugin

Release 1.0.1

Nov 14, 2020

1	Download the project and set up your environment	3
1.1	Step 1: Download the repository	3
1.2	Step 2: Choose an IDE	4
1.3	Step 3: Set your environmental variables	4
1.4	Step 4: Select the correct Python interpreter	5
1.5	Step 5: Access your git system from within your IDE	6
1.6	Step 6: Install python packages	6
1.7	Step 7: Build your resources file	6
1.8	Step 8: Start coding!	8
2	An overview of the project structure	9
2.1	doc: the documentation folder	9
2.2	qgisplugin: the actual plugin folder	10
2.3	tests: test classes for your code	10
2.4	other files	10
3	Write your code	11
4	Build a user interface	15
4.1	Build a GUI from scratch	15
4.2	Use the processing toolbox to have QGIS build the plugin for you	19
4.3	Create a command line interface for working outside of QGIS	21
5	Testing your code with unit tests	25
6	Write project documentation	29
6.1	Some concepts of reStructuredText	30
6.2	Building your documentation locally	31
6.3	Building your documentation on Read the Docs	32
7	Build a Python package	33
7.1	Build your package locally	33
7.2	Build your package automatically with each new version update	34
8	Build your QGIS plugin	35
8.1	Have a look at the code	35
8.2	Create your QGIS Plugin zip file	35

8.3	Installing your QGIS Plugin from file	36
8.4	Uploading your QGIS plugin to the official QGIS Plugin Repository	37
9	Updating your information	41
10	Example API	43
10.1	Python code	43
10.2	Command Line Interface	44
	Python Module Index	47
	Index	49

Get help in transforming your code to a QGIS plugin, but also:

- Write test classes for your code;
- Create online documentation (like what you are reading now);
- What are the requirements to upload your plugin to the official QGIS repository;
- Build and upload a python package to PyPi.

The **instruction pages** can be found at <<https://create-qgis-plugin.readthedocs.io>>.

The code **repository** can be found at <https://bitbucket.org/kul-reseco/create-qgis-plugin>.

PLEASE GIVE US CREDIT

When using this project as the base for your own plugin, please give us some credit in your acknowledgements like this:

The project structure is based on 'Create A QGIS Plugin' created by Crabbé Ann, Jakimow Benjamin and Somers Ben and funded by BELSPO STEREO III (Project LUMOS - SR/01/321). The full code is available from <https://bitbucket.org/kul-reseco/create-qgis-plugin>.

ACKNOWLEDGEMENTS

The creation of this project is funded primarily through BELSPO (the Belgian Science Policy Office) in the framework of the STEREO III Programme – Project LUMOS - SR/01/321.

The LUMOS logo was created for free at <https://logomakr.com>.

SOFTWARE LICENSE

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License (COPYING.txt). If not see www.gnu.org/licenses.

For issues, bugs, proposals or remarks, visit the [issue tracker](#).

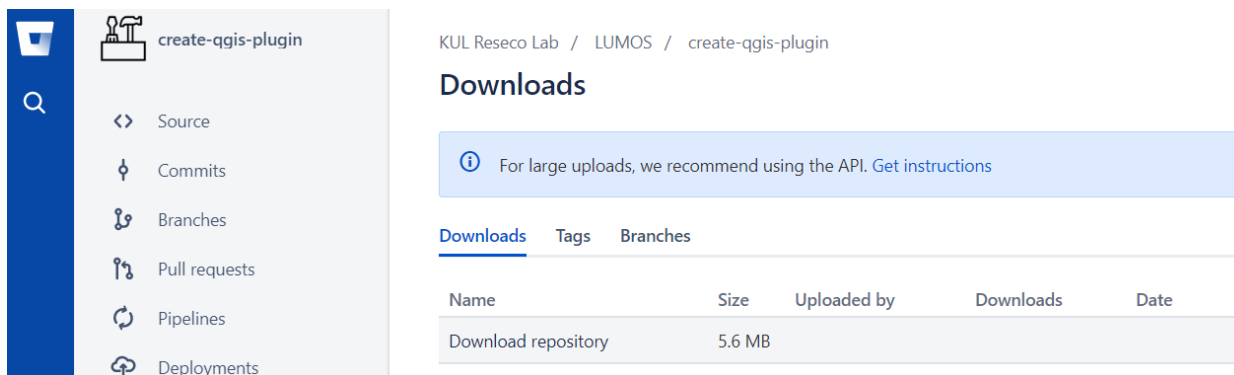


Download the project and set up your environment

For issues, bugs, proposals or remarks, visit the [issue tracker](#).

1.1 Step 1: Download the repository

You can find the [repository here](#). You can download the entire repository from the [downloads page](#).



The screenshot shows the GitLab interface for the repository 'create-qgis-plugin' under the path 'KUL Reseco Lab / LUMOS / create-qgis-plugin'. The left sidebar contains navigation links: Source, Commits, Branches, Pull requests, Pipelines, and Deployments. The main content area is titled 'Downloads' and includes a blue information box stating: 'For large uploads, we recommend using the API. Get instructions'. Below this, there are tabs for 'Downloads', 'Tags', and 'Branches'. The 'Downloads' tab is active, showing a table with one entry:

Name	Size	Uploaded by	Downloads	Date
Download repository	5.6 MB			

We recommend you use a versioning system like bitbucket, github or gitlab:

- create an account;
- create your own repository;
- clone it to your local system;
- copy our code into that folder.

From there, you can start changing the code and commit changes locally and/or push them to your remote system.

Now you are safe from system crashes! Not familiar with git? A lot of easy tutorials for starters are out there.

1.2 Step 2: Choose an IDE

Many **integrated development environments (IDE)** exists, like Microsoft Visual Studio, Spyder or Eclipse. An IDE supports you in writing, compiling and debugging your code, by helping you control your environment. Feel free to use the IDE of your choice, but keep in mind that set-ups tend to differ from IDE to IDE.

We are using **PyCharm**, as it is very powerful:

- Strong Python support;
- An intelligent editor that highlights incorrect syntax, has automated code-completion and docstring support;
- Smart code navigation allows you to jump to declarations in one click and helps your locate all usages of a function;
- Safe refactoring of code helps you to rename/delete variables and functions;
- Build-in debugging and testing;
- Build-in git (or SVN or Mercurial) support.

These two steps you will always have to do:

- Make sure your IDE used the correct python interpreter;
- Update your environmental variables, so that your interpreter knows where to look for qgis and osgeo packages.

Scroll down for an explanation on how to do this in PyCharm.

1.3 Step 3: Set your environmental variables

To run QGIS from python, your interpreter should know **where to go and look for the right packages**. Since these are no standard python packages, you will have to update your environmental variables.

You could try to do it manually, but we advise against it, as you could actually interfere with other processes on your computer and by installing new software your changes could be overruled.

The easiest way is to change them *on the fly* when you open your IDE:

- Create a batch file (e.g. *pycharm.bat*). It basically is a script with terminal commands;
- Copy/paste the code below;
- Lines with double colons (::) are comments;
- Edit the first two lines of code: they are specific to your QGIS installation;
- Save the file.

Now you can double click this batch file: Pycharm will open with in the background correct environmental variables.

```
::QGIS installation folder
set OSGEO4W_ROOT=C:\OSGeo4W64
set QGIS_PLUGINPATH="C:\Users\[...
↪]\AppData\Roaming\QGIS\QGIS3\profiles\default\python\plugins"

::set defaults, clean path, load OSGeo4W modules (incrementally)
call %OSGEO4W_ROOT%\bin\o4w_env.bat
call qt5_env.bat
call py3_env.bat
```

(continues on next page)

(continued from previous page)

```

:::lines taken from python-qgis.bat
set QGIS_PREFIX_PATH=%OSGEO4W_ROOT%\apps\qgis
set PATH=%QGIS_PREFIX_PATH%\bin;%PATH%

:::make PyQGIS packages available to Python
set PYTHONPATH=%QGIS_PREFIX_PATH%\python;%PYTHONPATH%

:: GDAL Configuration (https://trac.osgeo.org/gdal/wiki/ConfigOptions)
:: Set VSI cache to be used as buffer, see #6448 and
set GDAL_FILENAME_IS_UTF8=YES
set VSI_CACHE=TRUE
set VSI_CACHE_SIZE=1000000
set QT_PLUGIN_PATH=%QGIS_PREFIX_PATH%\qtplugins;%OSGEO4W_ROOT%\apps\qt5\plugins

::enable/disable QGIS debug messages
set QGIS_DEBUG=1

::PyCharm executable
set IDE="C:\Program Files\JetBrains\PyCharm Community Edition 2019.2.3\bin\pycharm64.
↪exe"
start "Start your IDE aware of QGIS" /B %IDE% %*

```

1.4 Step 4: Select the correct Python interpreter

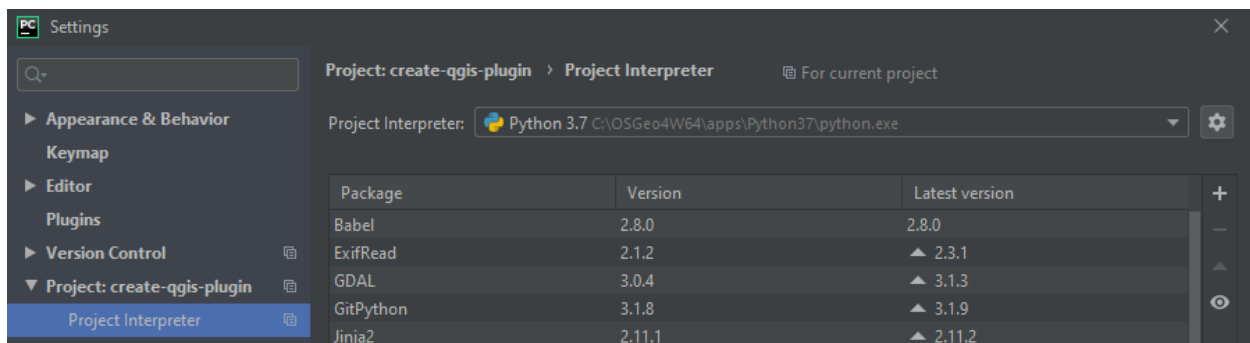
You probably will have, knowingly or unknowingly, installed many python versions on your system: stand alone, inside your QGIS installation, one for Microsoft Visual Studio, maybe when working in Spyder before, etc.

You have to choose the interpreter that comes with the QGIS installation. Usually it can be found here:

```
QGIS installation folder > apps > python37 > python.exe
```

Setting the python interpreter is straightforward:

- PyCharm will complain to you and you won't be able to run your code. Just follow the link in the warning message.
- Or go to File > Settings > Project > Project Interpreter.



1.5 Step 5: Access your git system from within your IDE

Add the git executable to your path by adding the following line of code to your batch file from step 3. Paste it *before* the last line of code where you call the program itself:

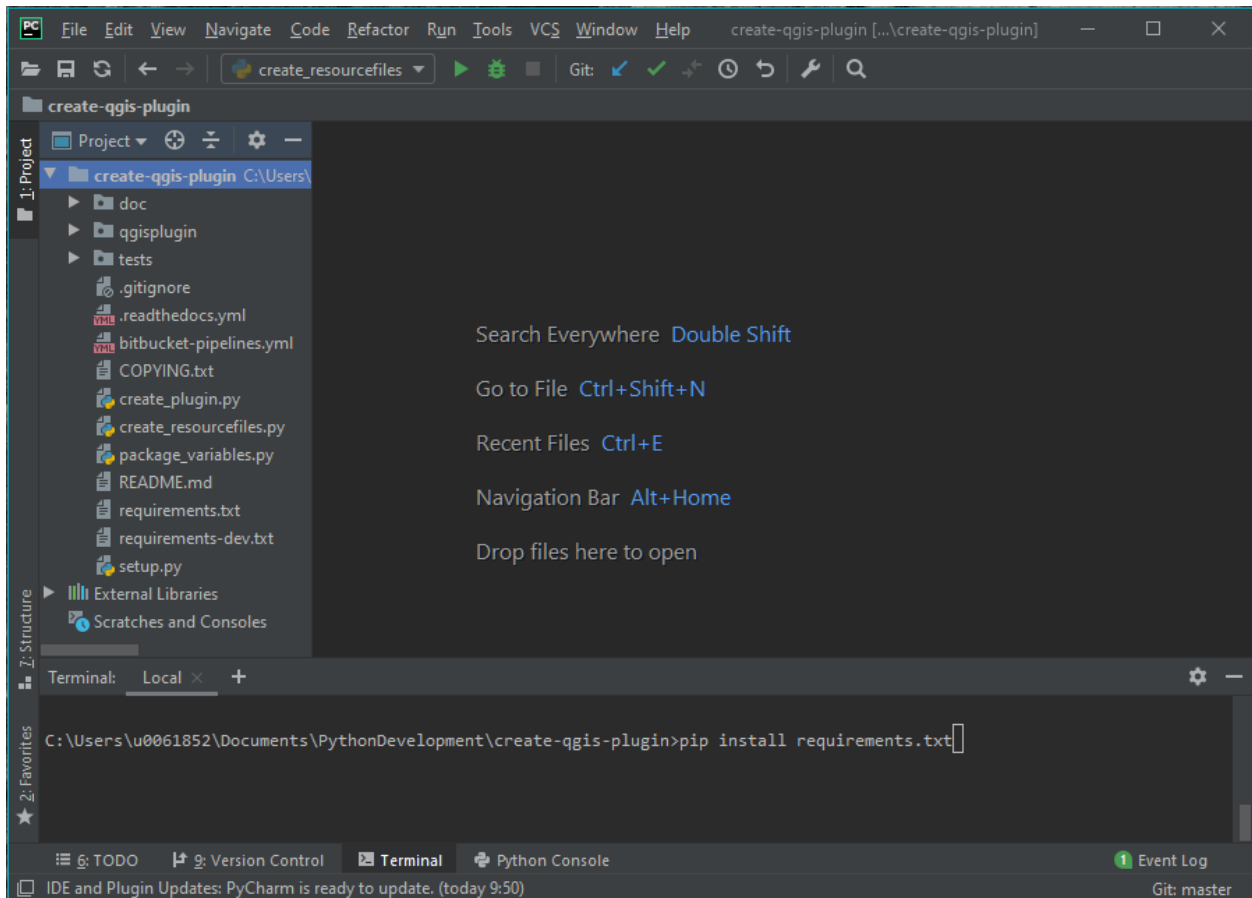
```
set PATH=%PATH%;C:\Users\[...]\AppData\Local\Programs\Git\bin
```

1.6 Step 6: Install python packages

The files *requirements.txt* and *requirements-dev.txt* contain all external packages that are important to your project. You can install them from within PyCharm:

- Go to the terminal window at the bottom of your project
- Type the following code:

```
$ pip install -r requirements.txt  
$ pip install -r requirements-dev.txt
```



1.7 Step 7: Build your resources file

If you want to access images from within python code (e.g. to add your logo to the GUI), you need a resource file.

In the *images* folder of your plugin, you will find one or more images, and a file named *resources.qrc*. This file contains a list of images, and where to find them, like this:

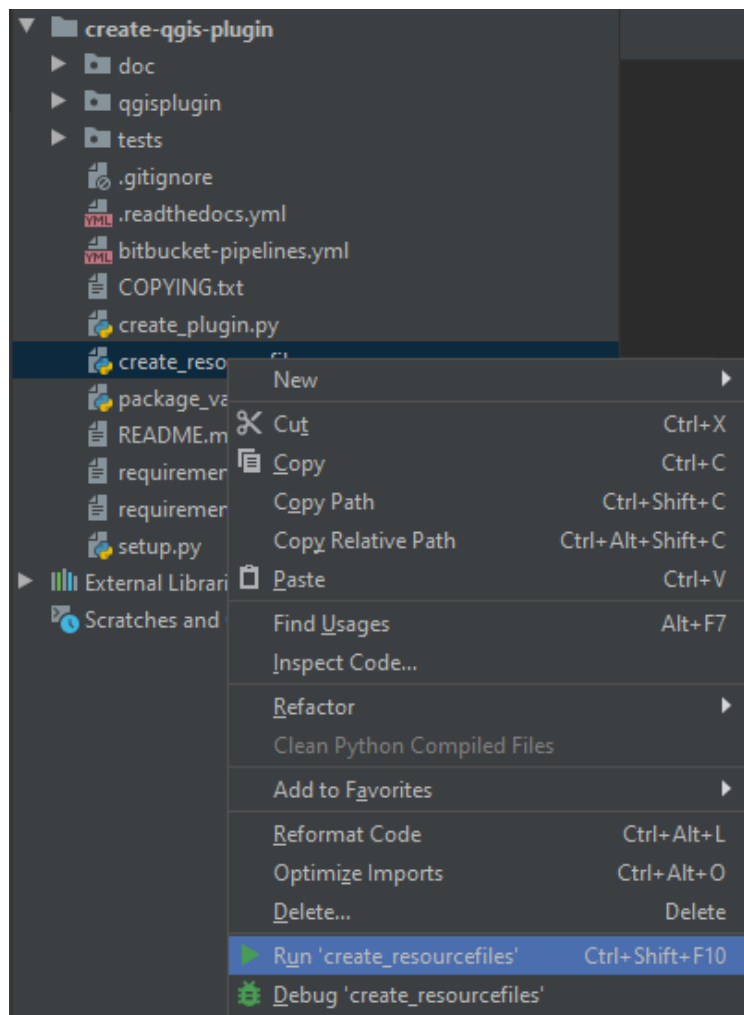
```
<RCC>
  <qresource>
    <file alias="plugin_logo">plugin_logo.png</file>
  </qresource>
</RCC>
```

Here, you can add a reference your own images if you would like:

```
<RCC>
  <qresource>
    <file alias="plugin_logo">plugin_logo.png</file>
    <file alias="new">my_new_image.png</file>
  </qresource>
</RCC>
```

This file is not enough. It must be translated into a python file. Luckily, we made that part easy for you:

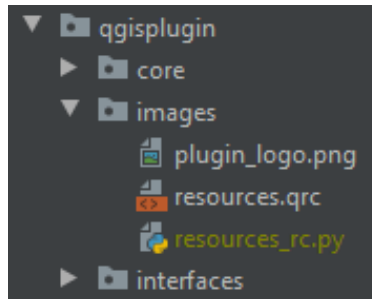
- Right-click on the file named *create_resourcefiles.py*.
- Choose *Run*.



This output should appear on the console

```
Recourse file 'C:\Users\[...]\create-qgis-plugin\qgisplugin\images\resources_rc.py'
↳ created.
```

And a new file should be added to the *image* folder:



We chose not to store this *resources_rc.py* file on git (see [the next page](#) for an explanation on *.gitignore*). That is why this step was necessary.

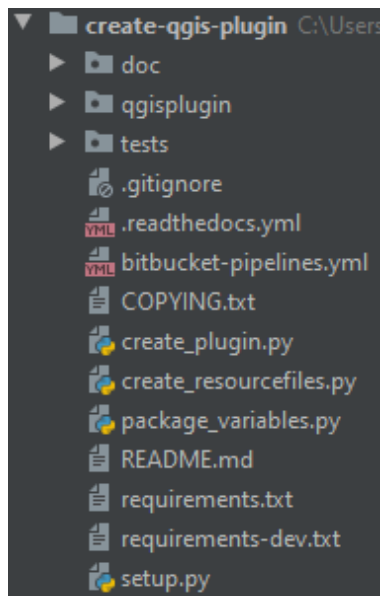
1.8 Step 8: Start coding!

The next chapters explain one by one how to build your code, interfaces, test classes, some documentation and of course the plugin itself!

Before you upload your code to an online platform, make sure you follow the steps [here](#) in order not to have incorrect licensing information or author identification.

An overview of the project structure

At first glance, you might be overwhelmed by the sheer number of files for such a small tool. Let's break it down and get you introduced to all parts. You will notice it is not that bad...



2.1 doc: the documentation folder

Here you will find all the files for building your documentation. It includes configuration files and content. For example, it includes the code for the page you are reading now. For more information, see [Part 4](#).

2.2 qgisplugin: the actual plugin folder

This is the most important folder in your project. It contains the actual code of your plugin. For more information, see [Part 1](#) on the code and [Part 2](#) on the user interface.

2.3 tests: test classes for your code

This folder should contain scripts for unit testing and test data. For more information, see [Part 3](#).

2.4 other files

The rest of the files are required for either building or maintaining your plugin.

- **.gitignore** tells git which files to ignore, so they won't clutter your repository.
- **.readthedocs.yml** contains metadata for the documentation hosting platform Read the Docs.
- **bitbucket-pipelines.yml** contains metadata to set up a pipeline on bitbucket.
- **COPYING.txt** contains the license information and should always be distributed with your code.
- The **create_plugin.py** script allows you to build your qgis plugin in one click.
- The **create_resourcefiles.py** allows you to create a resource file for your images in one click.
- **package_variables.py** contains recurring variables, like author name, version number, etc.
- **README.md** contains the basic project information and its content is used by your git page, readthedocs and PyPi.
- **requirements.txt** lists the external packages required to run your python core.
- **requirements_dev.txt** lists all external packages required for development.
- The **setup.py** script allows you to create a python package in one click.

CHAPTER 3

Write your code

We have already populated this project with a very simple plugin that allows you to add a constant to each pixel of an image and then set all pixels with a value below a given threshold to 0.

You will notice the code is organized like this:

```
> qgis plugin folder
  > core
  > images
  > interfaces
  my_plugin.py
```

This is done to keep functionality and interfacing separated:

- The **core** should be built unaware of file formats. It assumes basic parameters like numpy matrices, integers, booleans, etc as input and output variables.
- The **interfaces** deal with reading and writing files and user interactions (e.g. choosing a value).
- **my_plugin.py** adds a menu item to the *raster* menu and a new tool to the processing toolbox in QGIS.

We begin with discussing the **core** part of the code. The interfacing is discussed in [Part 2](#) and the building of the QGIS package is discussed in [Part 6](#).

Now have a look at the code and get familiar with the structure:

```
import numpy as np

class MyCode:

    def __init__(self, image: np.ndarray, normalize: bool = False, quotient: int = 255)

    def add_to_image(self, constant: float) -> np.ndarray

    def execute(self, constant: float, threshold: float, p=None, log: callable = print) -> np.ndarray
```

This text is no attempt to teach the syntax or even best practices of python. In the line of this project however, here are some importance practices for you to follow:

- Please document your code and do it correctly.
- You can give variable type hints in the definition of your function.

An example:

```
def my_function(self, parameter_name: float, other_parameter: bool=False) -> int:
    """
    Explain what your function does. Leave a blank line and then explain each
    ↪input
    and output parameter.

    :param parameter_name: Describe the first parameter
    :param other_parameter: Describe another parameter
    :return: describe the return value
    """
```

- If you want your QGIS plugin to have the same look and feel as standard tools, your widget should have a progress bar and log window. Even though ui is not of concern in the core of your code, you should provide a handle to pass on text messages, error codes and progress of your code. We have provided two variables for that. In case there is not ui, the text, error messages or progress will simply be printed in the terminal:

```
class MyCode:

    ...

    def execute(self, [...], set_progress: callable = None, log: callable = print)
    -> np.ndarray:
        """
        [...]

        :param set_progress: communicate progress (refer to the progress bar in
        ↪case of GUI;
                               otherwise print to console)
        :param log:
        ↪the GUI;
                               communicate messages (refer to the print_log tab in
                               otherwise print to the console)

        [...]
        """

        self.set_progress = set_progress if set_progress else printProgress
        self.print_log = log if log else print

        # example: use self.print_log to issue text output to the user
        self.print_log('Processing started ...')

        # example: use self.set_progress to change the progress bar
        self.set_progress(30)

        [...]

        return [...]

def printProgress(value: int):
    """ Replacement for the GUI progress bar """
```

(continues on next page)

(continued from previous page)

```
print('progress: {} %'.format(value))
```

Before you upload your code to an online platform, make sure you follow the steps [here](#) in order not to have incorrect licensing information or author identification.

Build a user interface

Several ways exist to interact with the user: you can build a GUI from scratch. This is the most labour intensive but has the highest flexibility. Much easier is to have your GUI built automatically by the processing framework. And a third option is to write a command line interface.

Before you upload your code to an online platform, make sure you follow the steps [here](#) in order not to have incorrect licensing information or author identification.

4.1 Build a GUI from scratch

Before you get started on building your own GUI, it might be worth your while to go on YouTube and watch a few videos on PyQt and PyQt for QGIS. That way you are more familiar with the concepts provided here.

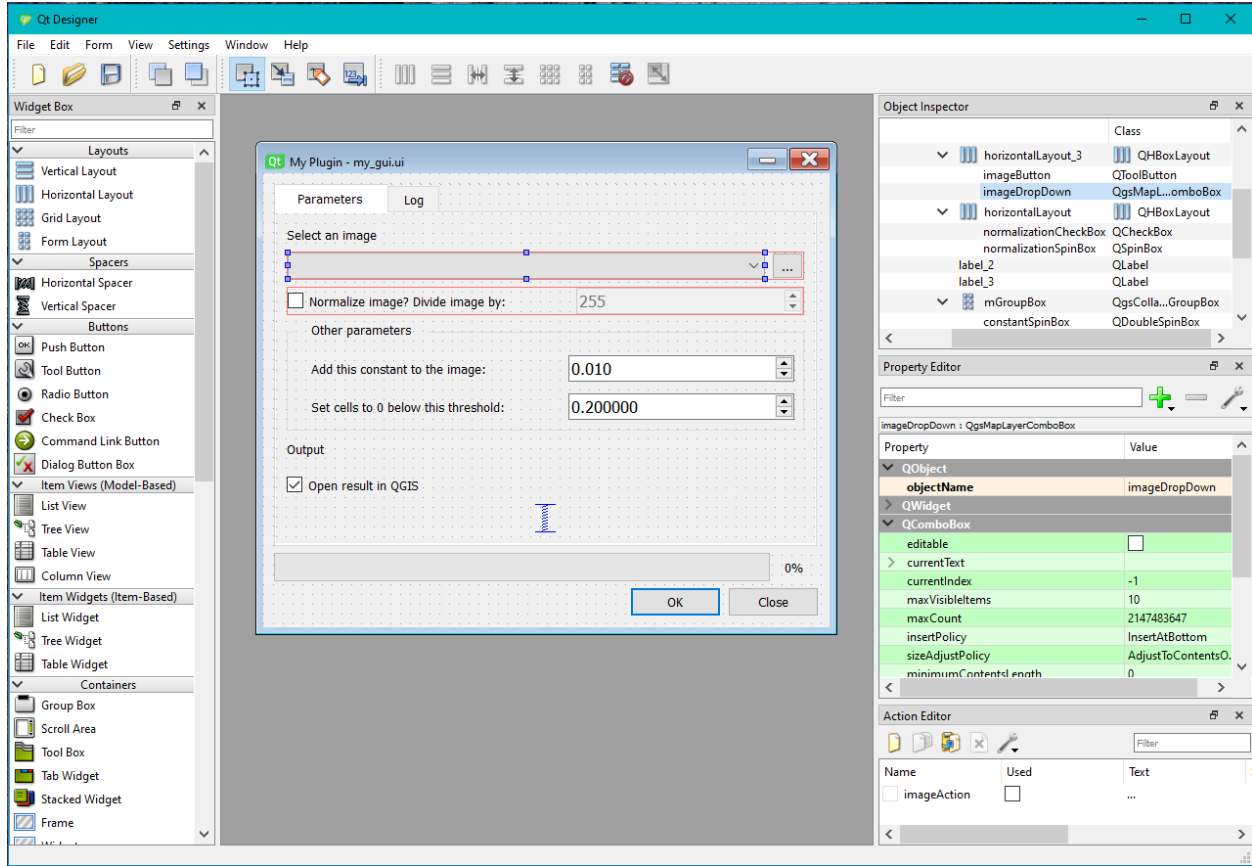
- Qt is a library to build widgets. PyQt is the Python package with Qt bindings. This library is independent from QGIS but QGIS uses it.
- Next to the Qt library, some QGIS specific widgets also exist.

To get started, we need two files:

```
> qgis plugin folder
  > images
  > interfaces
    my_gui.py
    my_gui.ui
```

The *.ui* file is an XML that describes the structure of your GUI. Do not try and edit it directly. You can open it in *Qt Designer*. This program is shipped with your QGIS installation and can be found under *QGIS installation folder > apps > Qt5 > bin > designer.exe* (see image below).

The the *.py* file connects the python code to the widgets found in the *.ui* file.

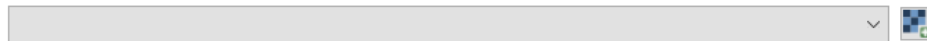


You can simply start from the GUI we have provided and edit it to make it suit your needs. Before you do, make sure you understand how all widgets work, and how they are connected with the code.

We give some important pointers here:

1. *Qt Designer* can only do so much. Some properties of your GUI should be set with code. For example, our GUI has a `QComboBox` to select an image from the QGIS table of contents and a `QPushButton` to look for an image on file.

Select an image



In the code snippet below, we make sure that (1) only valid raster layers can be selected; (2) the function `_choose_image` is called when the user chooses a layer from the `QComboBox`; (3) the function `_browse_for_image` is called when the user presses the `QPushButton`:

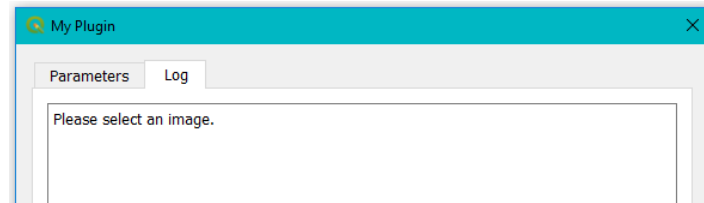
```
excluded_prov = [p for p in QgsProviderRegistry.instance().providerList() if p_
↳not in ['gdal']]
self.imageDropDown.setExcludedProviders(excluded_prov)
self.imageDropDown.setFilters(QgsMapLayerProxyModel.RasterLayer)
self.imageDropDown.layerChanged.connect(self._choose_image)
self.imageAction.triggered.connect(self._browse_for_image)
self.imageButton.setDefaultAction(self.imageAction)
```

2. In order to write output to the log screen in our widget, we created the `log` function. This function sends the text it receives to the `logBrowser` widget in our `.ui`. So instead of using `print()` statements, we now use `self.log()`.

```
def log(self, text: str):
    # append text to log window
    self.logBrowser.append(str(text) + '\n')
    # open the widget on the log screen
    self.tabWidget.setCurrentIndex(self.tabWidget.indexOf(self.tab_log))
```

It is this function, that is passed down to the core, in order to catch info and/or error messages:

```
result = MyCode([...]).execute([...], set_progress=self.progressBar.setValue,
↪log=self.log)
```

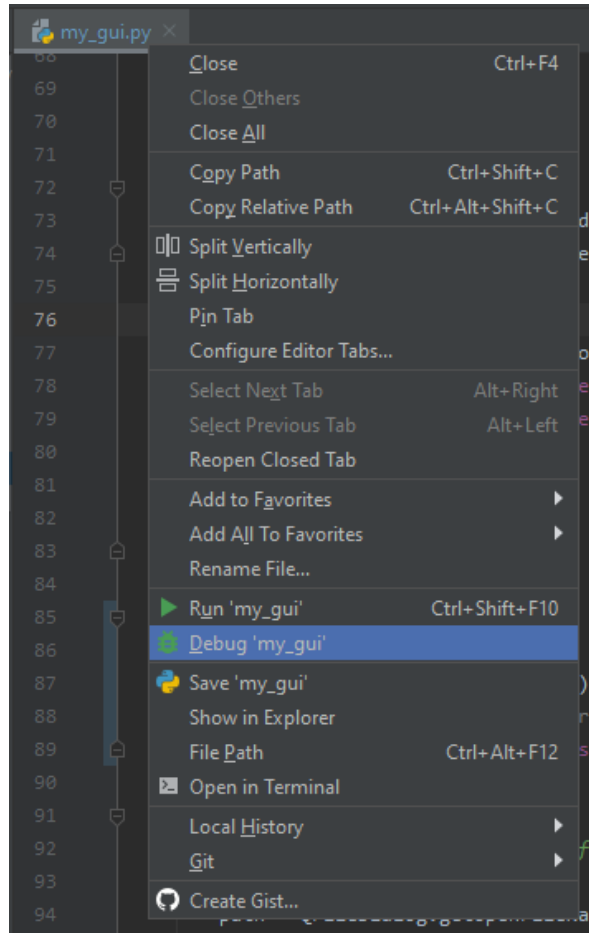


3. In a similar way the *setValue* function of the *QProgressBar* is passed down to the core (see code above).
4. To run your script without having to call it from a test class or open it in QGIS itself, you can simply add the following to the bottom of your script:

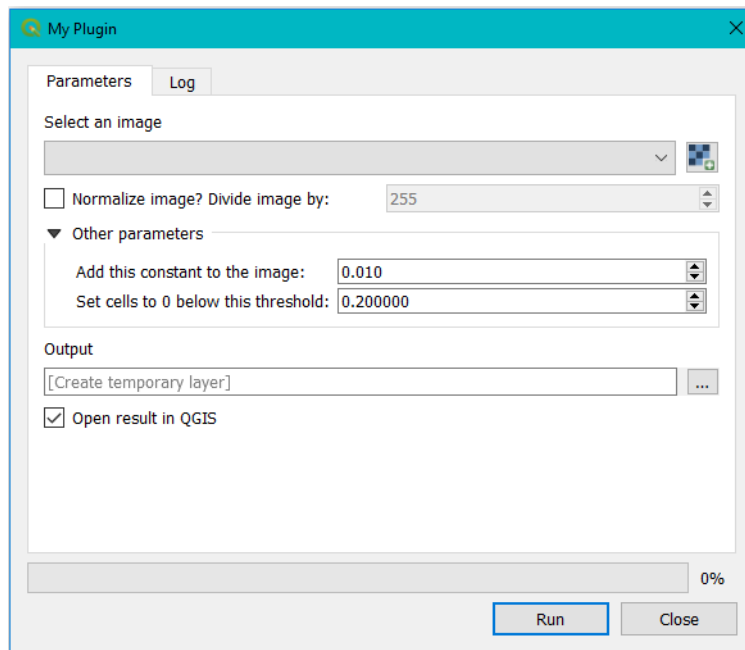
```
def _run():
    from qgis.core import QgsApplication
    app = QgsApplication([], True)
    app.initQgis()
    widget = MyWidget()
    widget.show()
    app.exec_()

if __name__ == '__main__':
    _run()
```

And then press *Run* or *Debug*:



The QGIS plugin looks like this:

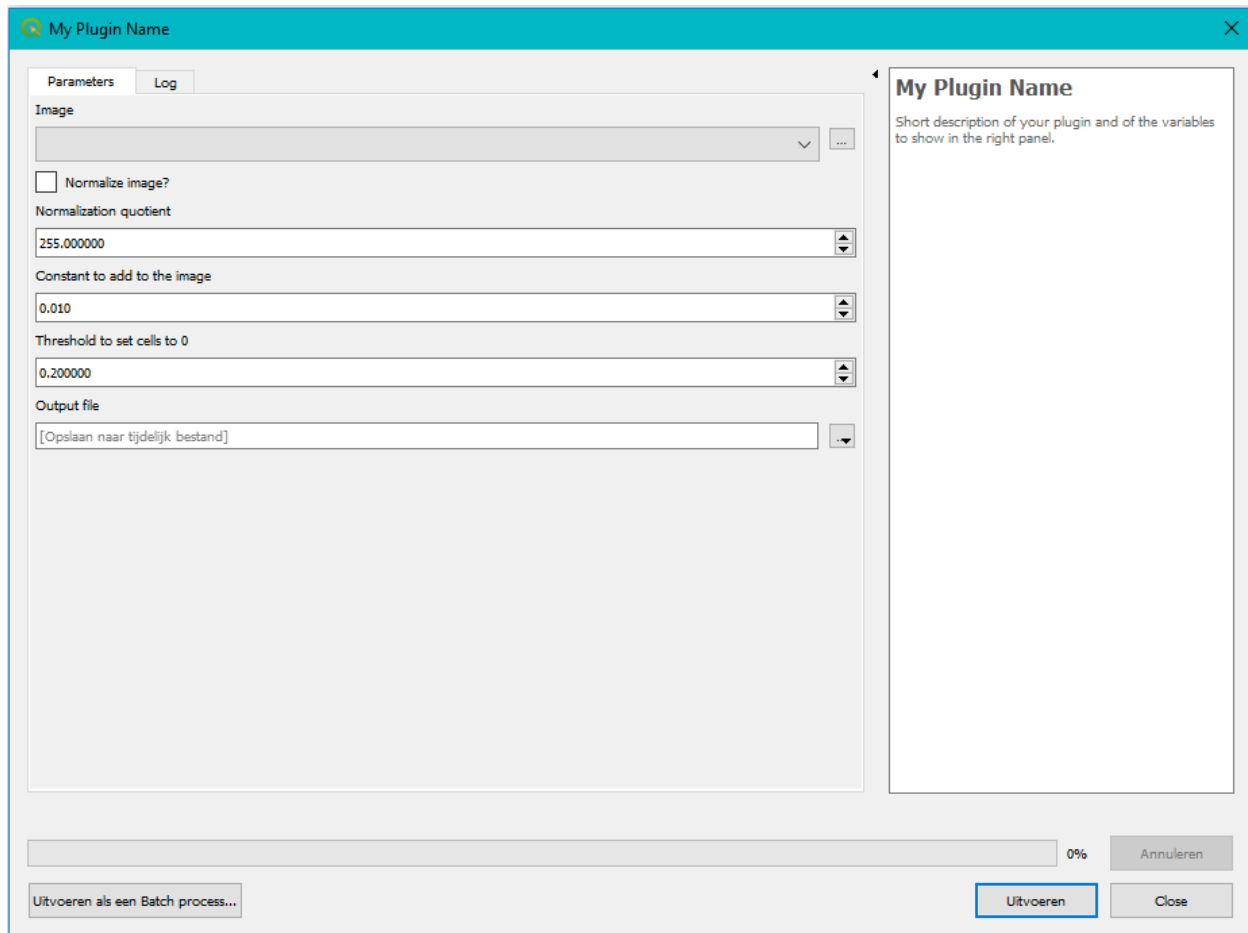


4.2 Use the processing toolbox to have QGIS build the plugin for you

You can use the QGIS Processing framework to have a plugin built **automatically**. This has a few advantages and disadvantages:

- [+] No worries about your widgets and/or lay-out. That is all taken care of in the background.
- [+] Your plugin can be executed in a batch process.
- [+] Your plugin can be part of the model builder.
- [-] You have less flexibility in placement of your widgets.
- [-] The widgets cannot interact dynamically.
- [-] Not possible to add to the menu bar. Your plugin can only be part of the processing toolbox.

Therefore it depends on the algorithm and the level of user interaction what is the best fit. You can always create both like we did.



Again you need two files:

```
> qgis plugin folder
  > images
  > interfaces
    my_plugin_processing.py
    my_plugin_provider.ui
```

Please analyze the scripts closely. There are a lot of obligated fields that are easy enough to understand like the name, display name, icon and helper string. All these fields are also accompanied with help in the documentation.

Two very important functions are *initAlgorithm* and *processAlgorithm*.

initAlgorithm describes all parameters that should be added to the widget: raster layers, numbers, booleans, ... Here you can specify the description of your parameter, a default value, min or max for numerical input, etc. Each input type has its own Qgs definition: *QgsProcessingParameterBoolean*, *QgsProcessingParameterNumber*, etc.

```
# Select a raster layer
self.addParameter(QgsProcessingParameterRasterLayer(
    name=self.INPUT, description=self.tr('Image'))
)

# Checkbox
self.addParameter(QgsProcessingParameterBoolean(
    name=self.NORMALIZE, description=self.tr('Normalize image?'), defaultValue=False)
)

# Numerical input
self.addParameter(QgsProcessingParameterNumber(
    name=self.NORMALIZATION_VALUE, description=self.tr('Normalization quotient'),
    ↪defaultValue=255,
    minValue=1, maxValue=1000000000, type=1)
)

# Select output file
self.addParameter(QgsProcessingParameterFileDestination(
    name=self.OUTPUT, description=self.tr('Output file'))
)
```

processAlgorithm is the function where you check all input parameters, transform them to the correct format and then call your core functionality.

```
def processAlgorithm(self, parameters, context, feedback):
    """
    Here is where the processing itself takes place.
    """
    # Read input values
    image_path = self.parameterAsRasterLayer(parameters, self.INPUT, context).source()
    image, metadata = import_image(image_path)
    normalize = self.parameterAsBoolean(parameters, self.NORMALIZE, context),
    quotient = self.parameterAsDouble(parameters, self.NORMALIZATION_VALUE, context)

    # Call the core function
    result = MyCode(
        image=image,
        normalize=normalize,
        quotient=quotient
    ).execute(
        constant=self.parameterAsDouble(parameters, self.CONSTANT, context),
        threshold=self.parameterAsDouble(parameters, self.THRESHOLD, context),
        set_progress=feedback.setProgress,
        log=feedback.pushInfo
    )
    result = result * quotient if normalize else result
    feedback.setProgress(100)
```

(continues on next page)

(continued from previous page)

```

# Write output to file
output_path = self.parameterAsFileOutput(parameters, self.OUTPUT, context)
output_path = write_image(file_path=output_path, image=result,
↳projection=metadata['projection'],
                        geo_transform=metadata['geo_transform'])

feedback.pushInfo("Written to file: {}".format(output_path))

# Add layer to your QGIS project
context.addLayerToLoadOnCompletion(output_path,
    QgsProcessingContext.LayerDetails(name='Processed Image', project=context.
↳project()))

return {'Processed Image': output_path}

```

4.3 Create a command line interface for working outside of QGIS

Building a command line interface is not that difficult, but you should execute it very carefully: it is very sensitive to typos.

Your script will have three parts:

1. A function that describes all mandatory and optional arguments. The code below shows 3 arguments, of which only the first one is mandatory:

```

parser.add_argument('image', type=str, help='Path to the input image.')
parser.add_argument('-t', type=float, default=0.2, help='Image threshold_
↳(default: 0.2).')
parser.add_argument('-o', type=str, help="Output file (default: 'output.tif")

```

2. A function to run the code based on the parsed arguments. This function is where you check all input parameters, transform them to the correct format and then call your core functionality.
3. A main function:

```

def main():
    parser = create_parser()
    run_code(parser.parse_args())

```

The tricky part is to get the command line function installed. The way to do this is to first set the function name in the *setup.py* file and point to the main function in your code:

```

entry_points={
    'console_scripts': [
        'myFunctionName=qgisplugin.interfaces.my_cli:main'
    ]
}

```

And then install your code as a python package. As long as you are still debugging, you can install it in *develop mode*: the files are not actually installed in the *site-packages* folder, but your system recognizes them as if it were. Go to the terminal window in your IDE and type:

```
$ python setup.py develop
```

From now on, you can call your package from the command line in the terminal like this:

```
$ myFunctionName images/sentinel2/august1.tif -t 0.01 -o images/results/august1_mask.  
↵tif
```

To get more info you use the `-help` or `-h` argument:

```
\create-qgis-plugin>mycli -h  
usage: mycli [-h] [-n N] [-c C] [-t T] [-o O] image  
  
todo: Give a short summary of what this code does here. e.g.: This script is a  
super simple example of a set of functions: one to multiply an image with a  
factor, one to add a constant to an entire image and one to set all values  
below a threshold to 0. This script only contains the mathematical part of  
your code and should be completely independent of i/o. You start from  
matrices, integers and other variables, and not from files or widgets!  
  
positional arguments:  
  image          Path to the input image.  
  
optional arguments:  
  -h, --help    show this help message and exit  
  -n N          To normalize your image, set the quotient here. (default: not  
                set)  
  -c C          Add this constant to the image (default: 0.01).  
  -t T          Threshold for setting cells to 0 (default: 0.2).  
  -o O          Output file (default: in same folder with name 'output.tif')
```

See [Part 5](#) on how to build python packages for distribution.

```

\create-qgis-plugin>mycli -h
usage: mycli [-h] [-n N] [-c C] [-t T] [-o O] image

todo: Give a short summary of what this code does here. e.g.: This script is a
super simple example of a set of functions: one to multiply an image with a
factor, one to add a constant to an entire image and one to set all values
below a threshold to 0. This script only contains the mathematical part of
your code and should be completely independent of i/o. You start from
matrices, integers and other variables, and not from files or widgets!

positional arguments:
  image      Path to the input image.

optional arguments:
  -h, --help  show this help message and exit
  -n N        To normalize your image, set the quotient here. (default: not
              set)
  -c C        Add this constant to the image (default: 0.01).
  -t T        Threshold for setting cells to 0 (default: 0.2).
  -o O        Output file (default: in same folder with name 'output.tif'
  
```

Testing your code with unit tests

Debugging and testing is very important when coding. Your IDE helps you a great deal by providing a debugger.

In addition, you should write test classes with so called *unit tests*: each *unit test* tests one aspect of your code. For example it runs your program with all default inputs and checks that the output is correct. Or it checks the behaviour of your CLI when the user inputs an invalid file. Or it checks that a specific button does what it is supposed to. Or ...

The point of writing these unit tests, is that you can run them quickly and frequently to check that your program is still doing what it is supposed to and does not break, even after you make some edits.

First you need test classes and test data. A class looks something like this:

```
import os.path as path
import unittest

from qgisplugin.core.my_code import MyCode
from qgisplugin.interfaces import import_image

DATA_FOLDER = path.join(path.dirname(__file__), "data")

class TestCore(unittest.TestCase):

    def test_core(self):
        # input
        image, _ = import_image(path.join(DATA_FOLDER, 'image.tif'))

        # run code
        result = MyCode(image=image, normalize=True, quotient=255).execute(constant=0.
↪01, threshold=0.2)

        # evaluate
        self.assertEqual(len(result), len(image))
        # todo it makes more sense to compare the actual content of the array, we_
↪leave this up to you
```

Next, you need a quick way to run your test classes. You have a few options.

You can run each class **manually**, if you only have a few: just right-click on each script and then click *Run*. You will get a summary on the console looking like this:

```

Run: pytest in manual_test_gui.py x
Tests passed: 2 of 2 tests - 3 s 988 ms
Test Results 3 s 988 ms
Testing started at 18:49 ...
C:\OSGeo4W64\apps\Python37\python.exe "C:\Program Files\JetBrains\PyCharm Community Edition 2019.2.3\helpers\pych
Launching pytest with arguments C:/create-qgis-plugin/tests/manual_test_gui.py in C:/create-qgis-plugin/tests

===== test session starts =====
platform win32 -- Python 3.7.0, pytest-6.1.1, py-1.9.0, pluggy-0.13.1 -- C:\OSGeo4W64\apps\Python37\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\create-qgis-plugin
plugins: cov-2.10.1
collecting ... collected 2 items

4 Run | TODO | Version Control | Terminal | Python Console
Tests passed: 2 (2 minutes ago)
    
```

You can run **all test classes at the same time with PyTest**. Run the following line of code in the terminal:

```
python -m pytest --cov qgisplugin.core --cov qgisplugin.interfaces --cov-report term-
missing tests
```

This command orders `pytest` to check all python scripts in the folder `tests` whose name begin with `test_` and run the unit tests within them. Alongside the failure or success rate of each test class, you get some statistics on the *coverage* of all scripts that are in the folders `qgisplugin.core` and `qgisplugin.interfaces` (see parameter `-cov`). This coverage tells us what % of the code is covered by the test classes. The parameter `term-missing` then adds more information: the lines of code that have not been tested by our test classes. It is difficult to reach 100 % coverage, however this helps you detect vulnerabilities in your test regime.

```

===== test session starts =====
platform win32 -- Python 3.7.0, pytest-6.1.1, py-1.9.0, pluggy-0.13.1
rootdir: C:\create-qgis-plugin
plugins: cov-2.10.1
collected 5 items

tests\test_cli.py .. [ 40%]
tests\test_core.py . [ 60%]
tests\test_gui.py .. [100%]

----- coverage: platform win32, python 3.7.0-final-0 -----
Name                                                    Stmts  Miss  Cover   Missing
-----
qgisplugin\core\__init__.py                             0      0   100%
qgisplugin\core\my_code.py                             26      0   100%
qgisplugin\interfaces\__init__.py                       32      8    75%   35, 45-46, 55-60
qgisplugin\interfaces\my_cli.py                        33      3    91%   73-74, 78
qgisplugin\interfaces\my_gui.py                       117     36    69%   94-108, 122-133,
↪ 145, 165,
                                                    172-178, 200-
↪ 207, 211
qgisplugin\interfaces\my_plugin_processing.py           49     49     0%   2-148
qgisplugin\interfaces\my_plugin_provider.py            13     13     0%   2-44
-----
TOTAL                                                    270    109    60%

===== 5 passed, 15 warnings in 6.18s =====
    
```

You can also have these test run **automatically by your versioning system**. For example Bitbucket has a continuous development implementation named *pipelines*.

In order to make use of this option, you need the *bitbucket-pipelines.yml* yml file. With the following implementation the test suite is run with each commit to the repository:

```
image: qgis/qgis

definitions:
  steps:
    - step: &Test-step
      script:
        - python -m pip install -r requirements.txt
        - python -m pip install -r requirements-dev.txt
        - python setup.py develop
        - apt-get update
        - apt-get install -y xvfb
        - apt-get install -y wget
        - apt-get install -y unzip
        - Xvfb :1 -screen 0 1024x768x16 &> xvfb.log &
        - ps aux | grep X
        - DISPLAY=:1.0
        - export DISPLAY
        - python create_resourcefiles.py
        - python -m pytest --cov qgisplugin.core --cov qgisplugin.interface --cov-
↪report term-missing tests

pipelines:
  branches:
    default:
      - step: *Test-step
    staging:
      - step: *PYPI-step
```

Note: The bitbucket pipeline mechanism seems to have trouble with testing the GUI, as it uses a timer to open and close widgets. Therefore we have taken it out of the equation by prefixing the script with *manual_*. This test will therefore not be part of any automated testing, but you can run it manually now and then to check that your GUI is still functioning properly.

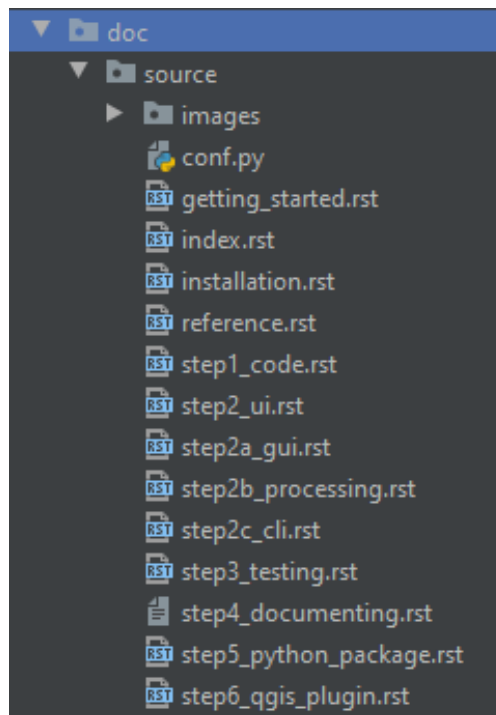
Before you upload your code to an online platform, make sure you follow the steps [here](#) in order not to have incorrect licensing information or author identification.

Write project documentation

Again it is best to familiarize yourself a little bit with the reStructuredText, Markdown, Sphinx and Read the Docs. You do not need to be a pro in any of these fields to build something that looks descent. But it might be overwhelming in the beginning.

You could start from scratch yourself. But we made it easy for you and presented this template for you. No need for you to install packages, configure sphinx or anything like that: all you have to do is deliver content.

The documentation folder looks like this:



- The *images* folder contains all images.

- The *conf.py* file contains configuration settings, but should not concern you at this point.
- The *index.rst* file is the first page the user will see, but all other *.rst* files contain the rest of the content. These pages are written as *reStructuredText*.

To write meaningful documentation, **please try to stick more or less to the following structure:**

1. Some context on the plugin
2. Installation instructions
3. A user manual
4. One or more exercises
5. The API

6.1 Some concepts of reStructuredText

Have a look at *index.rst* to get familiarized with some concepts.

We always choose to have Section 1 (context) be identical to the README.md file content. That way we make sure not to duplicate any information that might be lost or forgotten.

Our *README.md* contains information on the plugin, citations and acknowledgments, license, sponsors etc.: all information you want the user to know upfront.

To copy the information, use the *include* directive:

```
.. include:: ../../README.md
```

README.md is written in the *Markdown* format which is not entirely compatible with *reStructuredText*. Best keep it simple. Luckily headers work the same way.

Warning: Try not to mess with the *index.rst* and *README.md* structure: it is designed in such a way to have a nice lay-out of your HTML pages and at the same time have a good table of context and structure of the PDF version of your documentation. This is not evident when you start including other files like we do here.

You can build a table of contents using the ‘toctree’ directive:

```
.. toctree::
   :hidden:

   installation
   getting_started
   step1_code
   ...
```

The *:hidden:* parameter is to not have it duplicated on the *index.html* page.

Hyperlinks are created like this:

```
`issue tracker <https://bitbucket.org/kul-reseco/create-qgis-plugin/issues>`_
```

Images are added like this:

```
.. image:: images/gui_empty.PNG
   :width: 40 %
   :align: center
```

Another important part of documentation is **sharing your API**. This is done like this for a python class:

```
.. automodule:: qgisplugin.core.my_code
   :members:
   :undoc-members:
   :show-inheritance:
```

And like this for a command line interface:

```
.. argparse::
   :module: qgisplugin.interfaces.my_cli
   :func: create_parser
   :prog: mycli
```

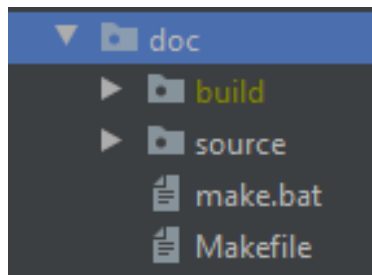
Just have a look at the rest of the documentation for more inspiration.

6.2 Building your documentation locally

To build your documentation locally, all you have to do is run this line of code in the terminal:

```
$ cd doc
$ make html
```

A new folder will appear next to the source folder:



To test your HTML pages, go to `doc > build > html > index.html` and right-click on the file. You will see an option to open the file in a browser. Your documentation will appear in your web browser the same way they would on the Read the Docs platform.

You will notice your browser tab has an **icon (favicon)**. To change this do the following:

- If you do not want to touch `conf.py`, copy your icon to the `images` folder and name it “plugin_logo.PNG”.
- Or you can find (and change) the reference to this icon in the `conf.py` file like this:

```
html_favicon = 'images/plugin_logo.PNG'
```

6.3 Building your documentation on Read the Docs

Before you upload your documentation to an online platform, make sure you follow the steps [here](#) in order not to have incorrect licensing information or author identification.

Then go to readthedocs.org to make an account and follow the easy steps to link your readthedocs account with your online repository. Just make sure you choose a clear name for your documentation and make sure it is the same as in your *package_variables.py* file.

And that is it! Updates will now happen automatically.

Build a Python package

Before you upload your python package to an online platform, make sure you follow the steps [here](#) in order not to have incorrect licensing information or author identification.

We have prepped the *setup.py* file in order to easily build the python package. It uses information from your *package_variables.py* file so be sure the content of this file is correct.

With each new release, make sure you **update the version number** in *package_variables.py*. Otherwise PyPi will complain that a package with this version number already exist and your upload will fail.

7.1 Build your package locally

Install twine if you haven't already:

```
> python -m pip install twine
```

Build the distribution:

```
> python setup.py sdist
```

Your package is automatically saved in a folder named 'dist'. Now you can upload the distribution to PyPi (you need a login first):

```
> python -m twine upload dist/your-package-name.tar.gz -u [PyPi username] -p [PyPi ↵  
↵password]
```

Your package is now available to everyone for installation using the pip command:

```
> pip install your-package-name
```

7.2 Build your package automatically with each new version update

Normally, you **release a new version** of your packages only when you have a new stable product. In a professional environment, developers usually work on the *dev* or *master* branch of the code repository. Only when you want to release a new version, you merge your changes to a separate branch (e.g. *staging*).

Why? Have a look at the *bitbucket-pipelines.yml* yml file

```
image: qgis/qgis

definitions:
  steps:
    - step: &PYPI-step
      script:
        # packaging for pip and uploading
        - python -m pip install twine
        - python setup.py sdist
        - python -m twine upload dist/*.tar.gz -u $PYPI_UNAME -set_progress $PYPI_PW

pipelines:
  branches:
    staging:
      - step: *PYPI-step
```

With this code, the package will be built and uploaded to PyPi automatically, each time you push your changes to the *staging* branch. If we would not split up this functionality per branch, you could not avoid pushing a new version to PyPi in the middle of your development, something we want to avoid for obvious reasons.

With each new release, make sure you **update the version number** in *package_variables.py*. Otherwise PyPi will complain that a package with this version number already exist and your upload will fail.

Notice how we **do not write our username and password in plain text**. Instead, we store them as repository variables on bitbucket: go to *Repository settings* > *Repository variables* > add your username and password as 'PYPI_PW' and 'PYPI_UNAME' (or choose another name). Now the bitbucket pipeline will know how to access your PyPi account.

Build your QGIS plugin

The central files for building your plugin are *my_plugin.py* and *__init__.py* next to it. These are the scripts where you add your plugin to the menu bar or processing toolbox (or both).

8.1 Have a look at the code

The function *initGui* is called upon installation of your plugin. At this point you add an “action” (= button) to your freshly created menu and you add your processing provider to the QGIS Processing Registry.

When you uninstall your app, the function *unload* is called. It is important to remove your menu item and processing provider completely at this point.

Notice how upon action, the widget is called from a separate function named *run*. This design is intentional. If you would call your widget directly with the callback parameter, your code is activated as soon as you install your plugin or later, as soon as QGIS boots. This might cause unwanted processing power overhead.

8.2 Create your QGIS Plugin zip file

Before you upload your QGIS plugin to an online platform, make sure you follow the steps [here](#) in order not to have incorrect licensing information or author identification.

Building a QGIS is a bit peculiar. For example it is important to have a *metadata.txt* file with the correct content. Next, you are working under the GNU General Public License and you should make sure you adhere to their specifications too:

- You need to add the *COPYING* file to your code.
- Each script should contain the license information and a copyright disclaimer.

We have taken care of this for you:

- The license/copyright information can be found at the top of each script and should be edited by you to include the correct information.

- The COPYING file is part of this distribution and can not be removed.
- The metadata file is written automatically based on the information provided by you in the file *package_variables.py*.

How to build your plugin?

- Right-click on the file *create_plugin.py*.
- Click *Run*.

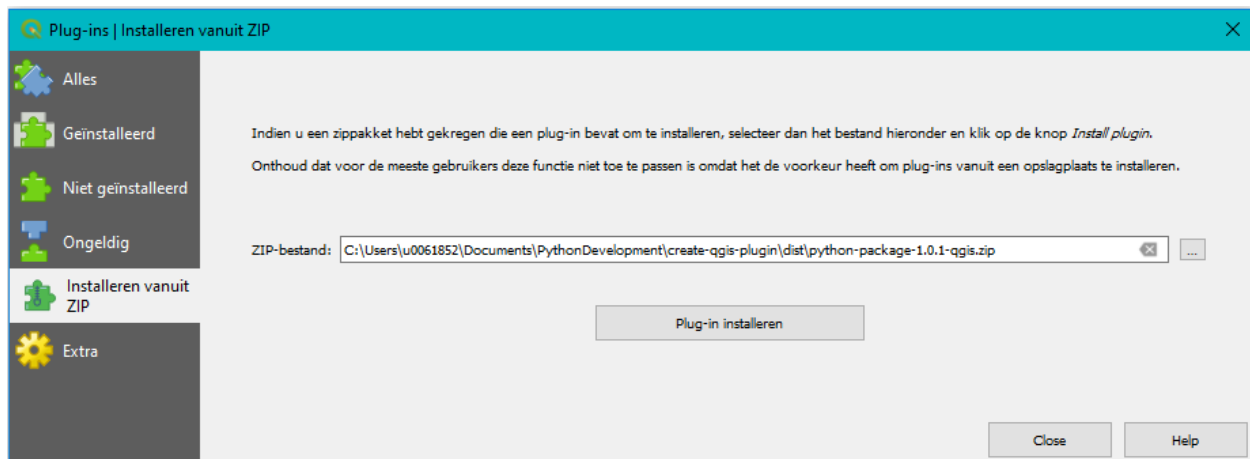
That is it! After this script has run successfully, you will find the following new files:

- *qgisplugin > images > resources_rc.py* (if you would not have done this already)
- *qgisplugin > metadata.txt*
- *dist > python-package-x.x.x-qgis.zip*

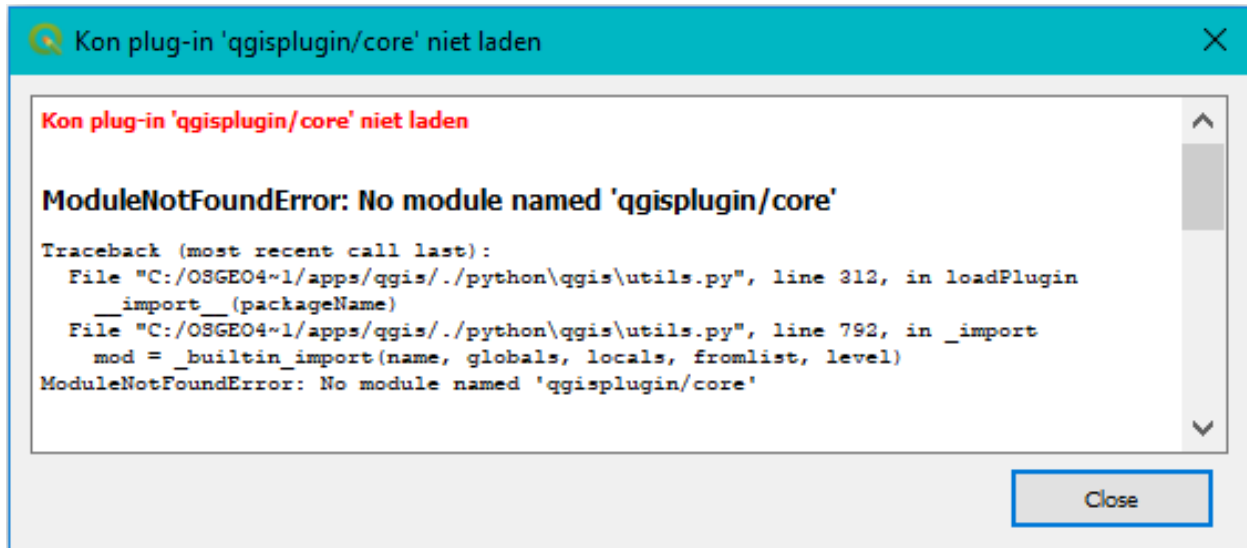
This last file is your QGIS plugin.

8.3 Installing your QGIS Plugin from file

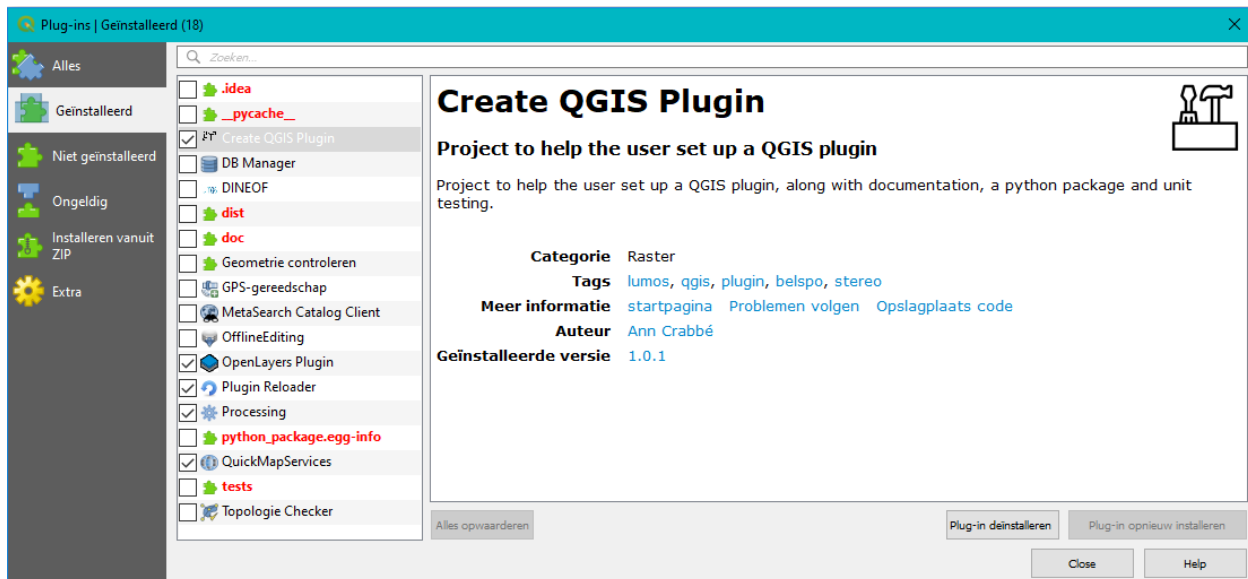
Your QGIS plugin can now be installed from file.



Do not be alarmed if you get an error message like this:



This happens because QGIS does not recognize the folder structure. But the plugin is installed regardless. Just turn your plugin once off and on in the *installed plugins* menu and you will be fine:



8.4 Uploading your QGIS plugin to the official QGIS Plugin Repository

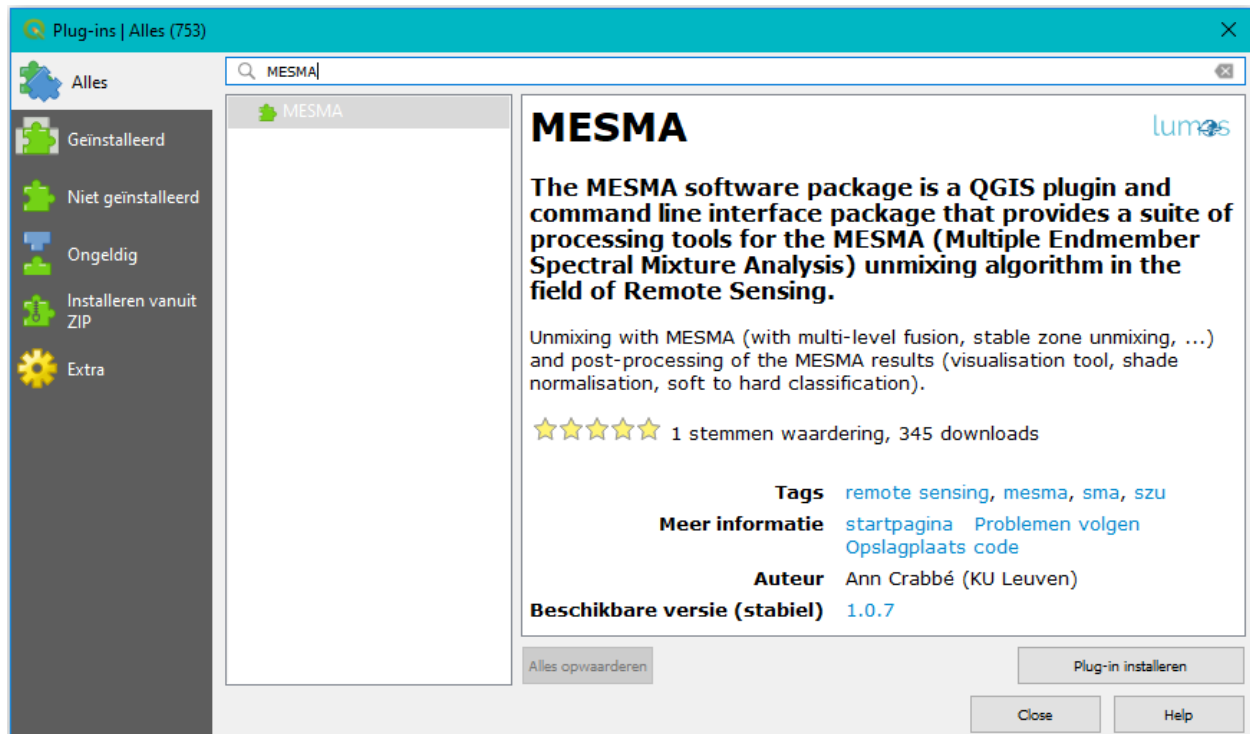
You can also upload your plugin to the official QGIS repository. In order to do that, you need an OSGEO account. Go to <https://plugins.qgis.org/plugins/my> and login with your account.

Here you can upload your zip file. A quick check is done on the content of your zip folder. If your metadata file is missing or corrupt, you will get an error message for example.

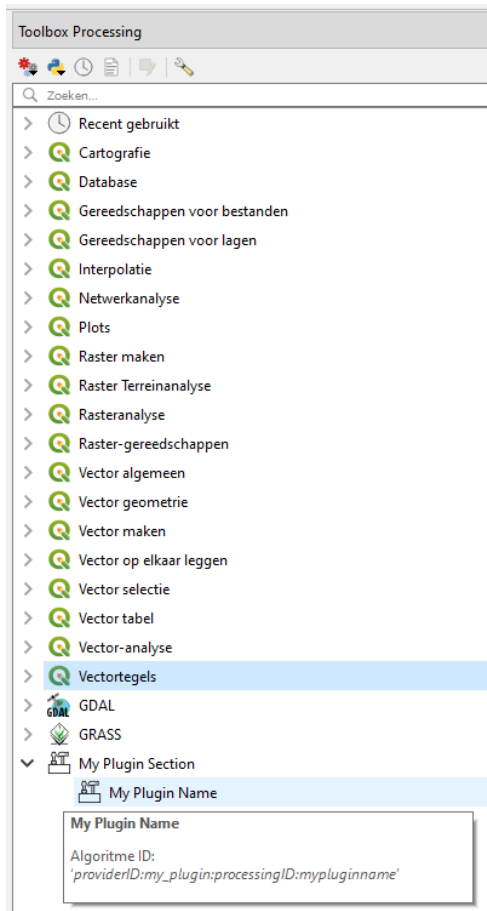
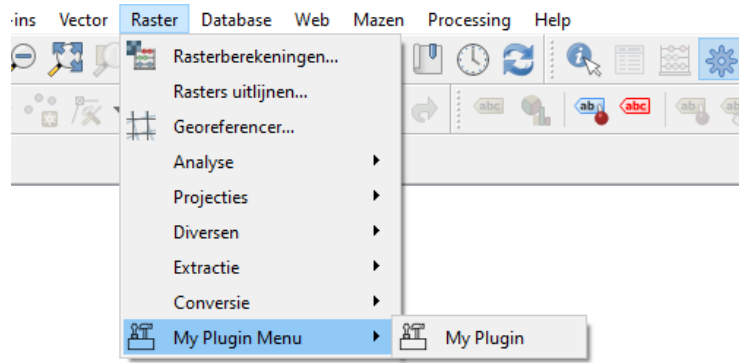
Then your plugin is checked manually by someone from the QGIS team and approved if they find that all conditions are met. Keep an eye on your issue tracker, as this is the first place they will post any comments, questions or problems.



Once your plugin has been approved, anyone can find and install it from within QGIS by typing the name or any of the keywords in the plugin search bar:



After installation, find your plugin in the menu or processing toolbox:



Updating your information

Before you go public with any product, whether it is the code, the documentation, the python package or the QGIS plugin, make sure you update a few key pieces of information throughout the code.

First of all, there are the variables in the file *package_variables.py*, like the plugin name, author, email, ...:

```
short_name = 'python-package'
qgis_plugin_name = 'Create QGIS Plugin'
read_the_docs_name = 'CreateQGISPlugin'
sphinx_title = 'How to create a QGIS plugin'

author = 'Ann Crabbé'
author_email = 'acrabbe.foss@gmail.com'
author_copyright = '@ 2020 by Ann Crabbé'
short_version = '1.0'
long_version = '1.0.1'

[...]
```

Also make sure you update all the code headers and replace our copyright and contact information with your own:

```
"""
| -----
| ↪ -----
| Date                : October 2020
| Copyright           : © 2020 by Ann Crabbé
| Email               : acrabbe.foss@gmail.com
| Acknowledgements   : Based on 'Create A QGIS Plugin' [https://bitbucket.org/kul-
| ↪ reseco/create-qgis-plugin]
|                    : Crabbé Ann and Somers Ben; funded by BELSPO STEREO III,
| ↪ (Project LUMOS - SR/01/321)
|
| This file is part of the [INSERT PLUGIN NAME] plugin and [INSERT PYTHON PACKAGE,
| ↪ NAME] python package.
|
```

(continues on next page)

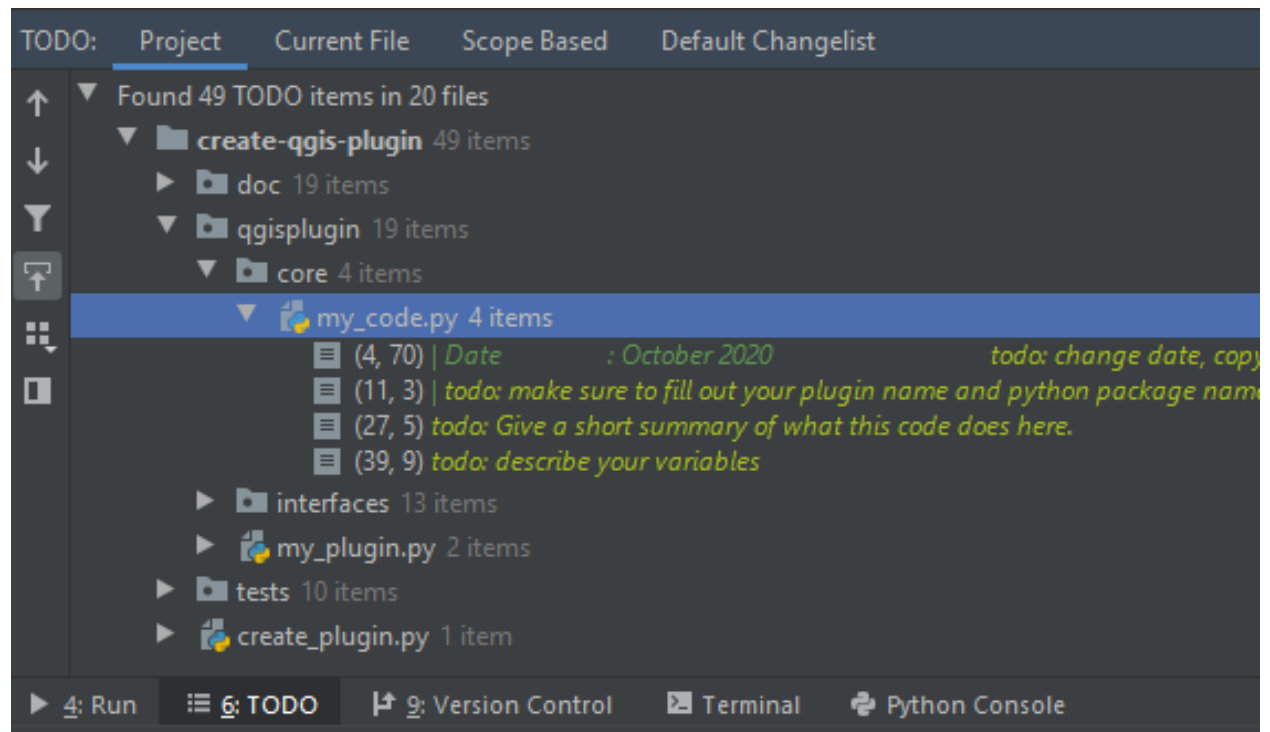
(continued from previous page)

```

| This program is free software: you can redistribute it and/or modify it under the
↳terms of the GNU General Public
| License as published by the Free Software Foundation, either version 3 of the
↳License, or any later version.
|
| This program is distributed in the hope that it will be useful, but WITHOUT ANY
↳WARRANTY; without even the implied
| warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
↳General Public License for more details.
|
| You should have received a copy of the GNU General Public License (COPYING.txt). If
↳not see www.gnu.org/licenses.
| -----
↳-----
"""

```

You will find *todo* statements all over the project. These indicated parts of the code or documentation that should be updated by you. To locate all of them easily, check the *todo* tab in PyCharm:



CHAPTER 10

Example API

This is an example of what your API will look like.

10.1 Python code

Date : October 2020 todo: change date, copyright and email in all files

Copyright : © 2020 by Ann Crabbé

Email : acrabbe.foss@gmail.com

Acknowledgements : Based on 'Create A QGIS Plugin' [<https://bitbucket.org/kul-reseco/create-qgis-plugin>]
Crabbé Ann and Somers Ben; funded by BELSPO STEREO III (Project LUMOS - SR/01/321)

This file is part of the [INSERT PLUGIN NAME] plugin and [INSERT PYTHON PACKAGE NAME] python package.

todo: make sure to fill out your plugin name and python package name here.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied

warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License (COPYING.txt). If not see www.gnu.org/licenses.

```
class qgisplugin.core.my_code.MyCode (image: numpy.ndarray, normalize: bool = False, quotient: int = 255)
```

Bases: object

todo: Give a short summary of what this code does here.

e.g.: This script is a super simple example of a set of functions: one to multiply an image with a factor, one to add a constant to an entire image and one to set all values below a threshold to 0.

This script only contains the mathematical part of your code and should be completely independent of i/o. You start from matrices, integers and other variables, and not from files or widgets!

```
add_to_image (constant: float) → numpy.ndarray
```

Add a constant to an image.

Parameters **constant** – The constant to add to each pixel of the image.

Returns The new image.

```
execute (constant: float, threshold: float, set_progress: callable = None, log: callable = <built-in function print>) → numpy.ndarray
```

This part is usually the core of your plugin: this function is called when the user clicks “run”.

Here we don’t do anything special: we add a number to an image and then set all values in an image to 0 where they are below a given threshold.

Parameters

- **constant** – The constant to add to each pixel of the image.
- **threshold** – all values below this threshold are set to 0
- **set_progress** – communicate progress (refer to the progress bar in case of GUI; otherwise print to console)
- **log** – communicate messages (refer to the print_log tab in the GUI; otherwise print to the console)

Returns the new image

```
qgisplugin.core.my_code.printProgress (value: int)
```

Replacement for the GUI progress bar

10.2 Command Line Interface

todo: Give a short summary of what this code does here.

e.g.: This script is a super simple example of a set of functions: one to multiply an image with a factor, one to add a constant to an entire image and one to set all values below a threshold to 0.

This script only contains the mathematical part of your code and should be completely independent of i/o. You start from matrices, integers and other variables, and not from files or widgets!

```
usage: mycli [-h] [-n N] [-c C] [-t T] [-o O] image
```

10.2.1 Positional Arguments

image

Path to the input image.

10.2.2 Named Arguments

- n** To normalize your image, set the quotient here. (default: not set)
- c** Add this constant to the image (default: 0.01).
Default: 0.01
- t** Threshold for setting cells to 0 (default: 0.2).
Default: 0.2
- o** Output file (default: in same folder with name 'output.tif')

q

`qgisplugin.core.my_code`, 43

A

`add_to_image()` (*qgisplugin.core.my_code.MyCode* method), 44

E

`execute()` (*qgisplugin.core.my_code.MyCode* method), 44

M

`MyCode` (class in *qgisplugin.core.my_code*), 43

P

`printProgress()` (in module *qgisplugin.core.my_code*), 44

Q

`qgisplugin.core.my_code` (module), 43