How to write a processor in Python

> **Under Construction**
> This page is still under construction and should be further improved.

Even though SNAP is based on the Java programming language it supports also the development of processor plugins which are written in Python. This is enabled by the so called snappy module. It enables Python developers to write processors for SNAP. In the background snappy usese the library jpy which builds a bridge between Java and Python.
This page shall guide you through the steps necessary for creating such a plugin. A working example of a Python processor plugin can be found in the snap-examples repository on GitHub. As you will use the Java API of SNAP Engine via snappy it is good to know this API. Before starting to implement an operator you should make sure that you have followed the procedure described in Configure Python to use the SNAP-Python (snappy) interface.

Project Setup

In order to write a Python plugin for SNAP you need the following tools.

- Maven >= v3.1
  Maven is the build tool used to build and package the final plugin file.
- 2.7 <= Python <= 3.4 (recommend is =3.3)

In order to make the development easier we recommend PyCharm as your IDE.

Project structure

The recommended basic project structure looks like as shown in the following code block. We will look at the content and the meaning of each of those files later.

Project Structure

```
<PROJECT_DIR>
|   pom.xml (the project definition file for maven)
\---src
    \---main
        +---nbm
        |       manifest.mf (settings for the plugin)
        +---python
        |       my_python_op-info.xml (metadata about the operator)
        |       my_python_op.py (the actual operator implementation)
        \---resources
            |   layer.xml (integration into SNAP Desktop)
            \---META-INF
                \---services
                        org.esa.snap.python.gpf.PyOperatorSpi (make the
operator known to SNAP Engine)
```

Operator implementation

To implement an operator you need to follow some conventions and rules. Have you heard already the saying "Don't call us, we call you!"?

That's how implementing implementing an operator for SNAP works. But here it has a positive meaning. Because you don't have to care for a lot of things you usually need to when developing an algorithm. Things like reading the input and writing the output. In SNAP you only need to do what you are asked for.

An operator implemented in Python needs to have three methods. An initialise, a compute and a dispose method. Please have a look at the following skeleton on an operator.

Operator skeleton

```
class MyOp:

    def __init__(self):
        pass

    def initialize(self, context):
        pass

    # You either implement this or the computeTileStack method
    def computeTile(self, context, band, tile):
        pass

    # You either implement this or the computeTile method
    def computeTileStack(self, context, target_tiles, target_rectangle):
        pass

    def dispose(self, context):
        pass
```

The **__init__** can be dropped if not necessary, but in most cases you will need. In the **initialize** method you do all the stuff which is necessary once in order to execute the algorithm, Things like preparing auxiliary data, validating the source product and so on. In the **dispose** m ethod you should release all resource (file handles etc.) which were allocated during the execution of your operator.
For the compute method you can choose between two variations. One, the **computeTile** method, is the one to be preferred in the general case. It computes target tiles independently from each other. The other, **computeTileStack** method, computes all tiles at once. This is useful if you get all results from a neural net at once or you have dependencies between your results.

Now you might ask your self what is meant by tile. A tile is a chunk of data. In SNAP data is processed in tiles. The full scene image is not processed at once. This would lead to a lot of memory problems. So we split the data into rectangular tiles. And each tile is processed individually. You are free to access any source data also outside of the target rectangle. You are just requested to fill the target tile rectangle with data. Where you get the data from is up to you.

The **context** object you see as a parameter to all of the methods gives you to some general useful methods.

Method	Description
`context.getSourceTile(rasterDataNode, rectangle)`	Retrieves the source data from the specified raster data node and the specified rectangle. (see API Doc)
`context.getSourceTile(rasterDataNode, rectangle, broderExtender)`	Retrieves the source data from the specified raster data node and the specified rectangle. If the rectangle exceeds the bounds of the source the ***borderExtend er*** defines how to handle this case. (see API Doc)
`context.getParameter('parameterName')`	Retrieves the value of the named parameter. (see API Doc)
`context.getSourceProduct()`	Get the source product which shall be processed. (see API Doc)
`context.getSourceProduct('sourceProductName')`	Get the named source product which shall be processed. (see API Doc)

By extending the rectangle used with one of the `getSourceTile` methods you can get data which is outside of the target region which is currently processed.

Operator Metadata

Each operator be accompanied by a xml file. This file must start with the same name as the module in which the operator is implemented and must end with **-info.xml**. This file is used to define processing parameters, the source product(s) and some more metadata about the operator like an operator alias name, description, version etc.

info.xml

```
<operator>
    <name>org.me.MyPyOp</name>
    <alias>my_py_op</alias>
    <operatorClass>org.esa.snap.python.gpf.PyOperator</operatorClass>
    <version>1.0</version>
    <authors>ACME Guys</authors>
    <copyright>(C) 2016 ACME</copyright>
    <description>
        A short description of the operator
    </description>
    <namedSourceProducts>
        <sourceProduct>
            <name>source</name>
        </sourceProduct>
    </namedSourceProducts>
    <parameters>
        <parameter>
            <name>factor</name>
            <description>Short description of the
parameter</description>
            <dataType>double</dataType>
            <defaultValue>1.0</defaultValue>
        </parameter>
    </parameters>
</operator>
```

A full example for such a xml file can be found in the example repository. In the following the meaning and the usage of the tags are described.

XML Tag	Description
`<name>`	A unique identifier within SNAP
`<alias>`	An user-friendly alias name to be used e.g. on the command line
`<operatorClass>`	This must be always org.esa.snap.python.gpf.PyOperator
`<version>`	The version of the operator. It should follow the concept of Semantic Versioning
`<authors>`	The names of the authors
`<copyright>`	The copyright notice
`<descrption>`	A short description of the operator shown on the command line

`<namedSourceProducts>`
`<sourceProduct>`
`<name>myInput</name>`
`</sourceProduct>`
`<sourceProduct>`
`<name>auxProduct</name>`
`</sourceProduct>`
`</namedSourceProducts>`
	This section defines one ore more source products. In the GUI currently one product is supported. On the command line it is possible to use the names along with the **`-S`** option to specify the source product, like **`-SmyInput=<path>`**

`<parameters>`
`  <parameter>`
`  <!--Parameter definiton-->`
`  <parameter>`
`<parameters>`
	In the parameters section all the parameters needed by the operator are specified. Each parameter is surrounded by the **`<parameter>`** tag.

Following the possible tags to be sued to configure a parameter (inside the **`<parameter>`** tag).

XML Tag	Description
`<name>`	The name of the parameter.
`<description>`	The description of the parameter shown on the command line and as tool-tip in the GUI.
`<label>`	Used in the GUI only instead of the name. The label can have white spaces in contrast to the name.
`<unit>`	The unit of the parameter. Shown in the the user interface.
`<dataType>`	The data type of the parameter, can be either **`byte,char,short,int,long,boolean,double,String`**.
`<defaultValue>`	The value this parameter shall have by default.
<notNull>	The parameter must be provided if set to **`true`**.
<notEmpty>	The parameter must not be empty if set to **`true`**. The difference to <notNull> is that even if the parameter tag present in a graph xml file it is not allowed to be empty. e.g. `<bandName></bandName>`
`<interval>`	The valid interval for numeric parameters, e.g. **`[10,20)`**: in the range 10 (inclusive) to 20 (exclusive)
`<valueSet>`	Set of values which can be assigned to a parameter field. The value set is given as textual representations of the actual values. (We have an issue with this property at the moment SNAP-730)
`<condition>`	A conditional expression which must return **`true`** in order to indicate that the parameter value is valid, e.g. **`value > 2.5`**
`<pattern>`	A regular expression pattern to which a textual parameter value must match in order to indicate a valid value, e.g. **`a*`**
`<format>`	A format string to which a textual parameter value must match in order to indicate a valid value, e.g. **`yyyy-MM-dd HH:mm:ss.Z`**
`<rasterDataNodeClass>`	The value can be **`org.esa.snap.core.datamodel.Band`** or any other fully qualified name of the sub-classes of RasterDataNode. This is useful to ensure that the user can only specify bands, tie-point grids or mask of the source product.

Graphical UI for the Operator

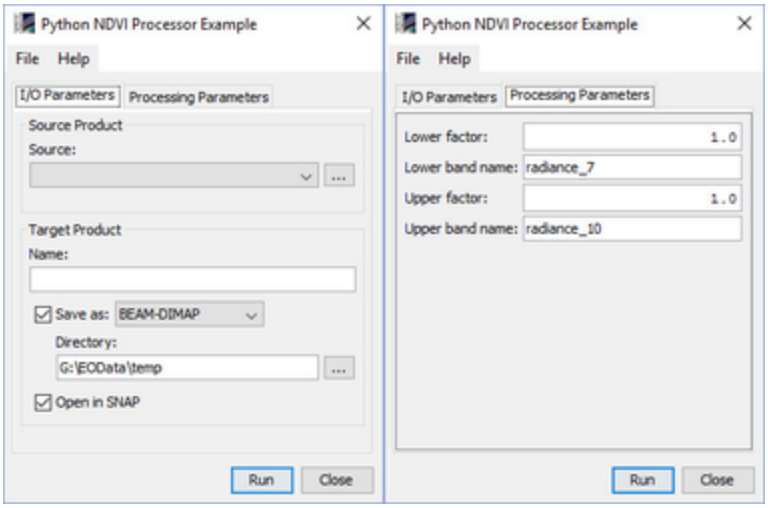
Additionally to the -info.xml file there is another xml file which is important if the operator shall have a graphical User Interface (GUI).

```
<filesystem>
    <folder name="Actions">
        <folder name="Operators">
            <file name="PyNdviOpAction.instance">
                <attr name="instanceCreate"
methodvalue="org.openide.awt.Actions.alwaysEnabled"/>
                <attr name="delegate"
methodvalue="org.esa.snap.core.gpf.ui.DefaultOperatorAction.create"/>
                <attr name="operatorName" stringvalue="py_ndvi_op"/>
                <attr name="displayName" stringvalue="Python NDVI
Processor Example"/>
                <attr name="dialogTitle" stringvalue="Python NDVI
Processor Example"/>
                <attr name="ShortDescription" stringvalue="Generates
NDVI from a source product with at least two spectral bands."/>
                <attr name="targetProductNameSuffix"
stringvalue="_ndvi"/>
                <attr name="helpId" stringvalue="pyNdviDoc"/>  <!--The
helpId if help contents is provided-->
            </file>
        </folder>
    </folder>
    <folder name="Menu">
        <folder name="Optical">
            <folder name="Examples">
                <file name="PyNdviOpAction.shadow">
                    <attr name="originalFile"
stringvalue="Actions/Operators/PyNdviOpAction.instance"/>
                    <attr name="position" intvalue="0"/>
                </file>
            </folder>
        </folder>
    </folder>
</filesystem>
```

This file defines the action which will invoke the GUI of the operator and it also defines the location of the action within the menu. The *Actions* fold er must contain another folder named *Operator*. Inside this the operator action and its attributes are defined. As a collector for the attributes a file is created and this file gets a name. You are free to choose any name here. But it should not interfere with others and it should end with the suffix **`.instance`**. The attributes **`instanceCreate`** and **`delegate`** should currently not be of any interest. They should always have the same value as in the example. The attribute **`operatorName`** is used to specify which GUI of which operator shall be called. This value must be the same as the **`alias`** specified in the info.xml file. The `displayName` defines the text of the menu entry and **`dialogTitle`** defines, as the name indicates, the title of the dialog. The attribute **`ShortDescription`** is a short description of the operator and used as tool-tip. If not other specified by the user the target product files will get the suffix specified by **`targetProductNameSuffix`**. This suffix is prepended before the file extension. The last attribute is the **`helpId`**. This is the id of the help page which shall be shown when the help button is pressed. Later more about this.

In the folder **`Menu`**, the menu structure of SNAP is reproduced. For each folder in the menu you create also a folder here with the same name. The final menu entry is then represented by a file which refers to the file defined above in the **`Actions`** section. Here it should end with the suffix **`.sh adow`**. The **`originalFile`** attribute links to the above defined action by using the full path to it. The **`position`** attribute defines the position of the entry within its parent folder.

Based on the definitions in the info.xml file a generic GUI is created for the operator when the defined menu entry is clicked. This comes right out of the box and without additional GUI layouting.



Currently it is not possible to define your own GUI. You have to stick to the GUI which is generated for you.

Provide User Help Pages

Oracle describes in this PDF document how to write help pages.

Known issues

- HDF library access
- ensure same Python configuration (used libraries) on the user system as the developer uses
- GPF.getDefaultInstance().getOperatorSpiRegistry().loadOperatorSpis() needs to be called
- Import-Vector can't be used (see forum thread)
- logging configuration might be changed by snappy (see forum thread)

Still TODO

- Write separate page on how to write help pages
- describe manifest.mf file
- pom.xml should be described
- PyOperatorSpi needs to be explained