

# HSL3 SDK documentation

## Version history

Version	From firmware version	Description/changes	Date
1.0	4.13.0	Initial release	23/01/2026

## Content

Version history.....	1
HSL3 SDK documentation.....	3
Introduction.....	3
Discontinuation of HSL2 .....	3
Disclaimer .....	3
Development environment.....	4
Setup: .....	4
Use: .....	4
First steps.....	5
Step 1: Create source code file.....	5
Step 2: Code structure.....	5
Step 3: Generate node.....	6
Step 4: Transfer .....	7
Classes .....	8
LogicModule.....	8
Hsl3Framework.....	9
Methods .....	9
Hsl3DebugSection .....	12
Methods .....	12
Hsl3Slots.....	13
Access .....	13
Methods .....	13
Hsl3Slot.....	15
Attribute.....	15
Libraries .....	15
External libraries .....	15

HSL3 generator .....	16
Overview .....	16
Conditions .....	16
Generator call up .....	16
Example.....	16
Description of command line parameters.....	17
Configuration file.....	17
Attributes of the configuration file .....	17
HSL 3 - Communication between logic node instances .....	20
Methods in the framework.....	20
Example .....	21
Process of communication.....	21
Notes on migration from HSL2 to HSL3.....	22
Introduction.....	22
Differences between HSL 2 and HSL 3.....	22
Tips for migrating logic nodes from HSL 2 to HSL 3.....	22
Converting Python 2 to Python 3 source code with 2to3 .....	22
Strings and bytes.....	23
Use of retentive memory variables .....	24

## HSL3 SDK documentation

### Introduction

As of firmware version 4.13 (Python version 3.9), it is now possible for users to develop their own logic modules for the Gira HomeServer or Gira FacilityServer using the HSL3 SDK.

This document describes how to develop logic nodes with the HSL3 SDK. The basic principles of logic node development are described in the HSL1 SDK.

In future firmware versions of the Gira HomeServer, there will be an increase in the Python version used (currently 3.9). Therefore, HSL3 logic nodes always need to be checked for compatibility with newer firmware versions. It is therefore advisable to ensure that no discontinued Python functions are used when creating logic nodes.

### Discontinuation of HSL2

The HSL2 SDK will no longer be supported in future versions of the firmware. Nodes based on the HSL2 SDK must be migrated to the HSL3 SDK.

### Disclaimer

The creator is responsible for the creation and function of logic nodes. This documentation is provided as an aid. Even if the documents have been prepared and checked with great care, errors cannot be completely ruled out. Therefore, the distributor assumes no legal responsibility and makes no guarantees as regards the content.

The right to make technical changes is reserved.

## Development environment

### Setup:

To set up a Python development environment suitable for the version, please follow the instructions below:

1. Install the Python install manager tool from <https://www.python.org/downloads/windows/>
2. Start Command Prompt (cmd.exe) and execute the following commands:

```
C:\Users\Your Name> py install 3.9
C:\Users\Your Name> cd Projekte
C:\Users\Your Name\Projekte> py -3.9 -m venv hsl3
C:\Users\Your Name\Projekte> cd hsl3
C:\Users\Your Name\Projekte\hsl3> Scripts\activate
(hsl3) C:\Users\Your Name\Projekte\hsl3>
```

3. Unzip the contents of the file "hsl3\_generator\_und\_beispiele.zip" into the folder (Projekte\hsl3).
4. Create an additional folder for each logic node project in which to store the source code.

### Use:

1. Start Command Prompt (cmd.exe)

```
C:\Users\Your Name> cd Projekte\hsl3
C:\Users\Your Name\Projekte\hsl3> Scripts\activate
(hsl3) C:\Users\Your Name\Projekte\hsl3>
```

2. The generator supplied can be used to generate a logic node from the code.
3. The generated \*.hsl file can then be imported and used with Expert version 4.13.0

```
(hsl3) C:\Users\Your Name\Projekte\hsl3> python generator3.cpython-39.pyc
--source examples\binary_trigger\config_binary_trigger.json
--target 10700_binary_trigger.hsl --debug
```

and above.

For more information, see Section entitled HSL3 Generator.

## First steps

Each logic node is based on a class with the name LogicModule. During instantiation, an instance of the Hsl3Framework class is transferred to the constructor.

### Step 1: Create source code file

Each file must begin with the prefix hsl3\_ and the node ID and end with the suffix .py. Only files with this structure can be converted by the HSL3 Generator.

As an example, let's use 10001 as the node ID. In this case, the node ID and freely enterable additional information "my\_module" leads to a source code file name hsl3\_10001\_my\_module.py.

### Step 2: Code structure

Once the empty file has been created, the script can be filled with the following basic structure:

```
LogicModule class:
    def __init__(self, hsl3):
        self.fw = hsl3

    def on_init(self, inputs, store):
        pass

    def on_calc(self, inputs):
        pass

    def on_timer(self, timer):
        pass
```

Each logic node always consists of the class LogicModule.

## Step 3: Generate node

With the help of the HSL3 Generator located in the SDK, the node can be converted into an HSL file.

### Step 3.1: Create configuration file

The configuration file is used to create the outer shell of the node. Here, a configuration file with the following content can be created under the file name config.json:

```
{
  "module": {
    "id": "10001",
    "context": "hsl3.examples",
    "category": "Examples",
    "name": "My first module",
    "hsl_filename": "10001_my_module.hsl",
    "inputs": [
      {
        "type": "number",
        "identifier": "In1",
        "init_value": 0,
        "label": "Input 1"
      }
    ],
    "outputs": [
      {
        "type": "string",
        "identifier": "Out1",
        "init_value": "-",
        "label": "Output 1"
      }
    ],
    "store": [
      {
        "type": "string",
        "identifier": "Sto1",
        "init_value": "-",
      }
    ],
    "timer": [
      {
        "identifier": "Tim1",
      }
    ]
  },
}
```

```
"scripts": [  
  {  
    "filename": "hsl3_10001_my_module.py"  
  }  
]  
}  
}
```

This simple configuration creates a node with an input and an output.

Step 3.2: Run the generator

Call up the generator in the configuration file folder:

```
python generator3x.pyc -source=config.json
```

This generates the desired HSL file.

Step 4: Transfer

The generated HSL file can be used directly with the HS/FS Expert and loaded directly on the device or on an HS/FS VM for quick testing.

## Classes

### LogicModule

The methods described below represent the basic framework of a logic node. If a project consists of several files, the LogicModule class may only appear in the main script.

#### Constructor

`__init__(self, hsl3fw)`

Must be specified and is called up when the node is instantiated.

The code is called up in the general context of the HSL3 environment. No blocking calls may be made here. Each delay affects all HSL3 nodes.

#### Parameters:

- hsl3 - Transfers an instance of the *Hsl3Framework* class.

#### Methods

`on_init(self, inputs, store)`

Called during logic initialisation. The code is listed in the context of the node.

#### Parameters:

- inputs - Contains all input values. Instance of the class Hsl3Slots.
- store - Contains the values of all memory variables. Instance of the Hsl3Slots class.

`on_calc(self, inputs)`

Is called if an input has been written and the node is recalculated.

#### Parameters:

- inputs - Contains all input values. Instance of the class Hsl3Slots.

`on_timer(self, timer)`

Called when a timer has been triggered. If no timers are used, this method can be omitted.

#### Parameters:

- timer - Contains all defined timers and their status. Instance of the class Hsl3Slots.



## Hsl3Framework

An instance of this class is transferred to each node during instantiation.

### Methods

`set_output(index_or_key, value)`

Sets the specified output to the transferred value. May only be called in the context of the node. If the method is called in a different context/thread, an exception is triggered.

#### Parameters:

- `index_or_key` - Index or key of the input. If the index or key does not exist, an exception is triggered.
- `value` - Value to be written to the input. Multiple calls in succession overwrite the previous value. Only one value is sent to the output.  
Important: The parameter must be of the type float, int or bytes. The HS/FS firmware only supports strings of the type "bytes" within the processing of the logic. De-/encoding may have to be performed by the node itself.

#### Exceptions:

- If the method is not called from the thread of the context, an exception of type `Hsl3ContextError` is triggered.
- If, for the value parameter, `None` or a "string" type value is transferred, an exception of the `ValueError` type is triggered.

`set_store(index_or_key, value)`

Sets the specified memory variable to the transferred value. May only be called in the context of the node.

#### Parameters:

- `index_or_key` - Index or key of the memory variable. If the index or key does not exist, an exception is triggered.
- `value` - Value to be written to the memory variable. Must be of the type float, int or bytes, otherwise an exception is triggered.  
Important: The parameter must be of the type float, int or bytes. The HS/FS firmware only supports strings of the type "bytes" within the processing of the logic. De-/encoding may have to be performed by the node itself.

#### Exceptions:

- If the method is not called from the thread of the context, an exception of type `Hsl3ContextError` is triggered.
- If, for the value parameter, `None` or a "string" type value is transferred, an exception of the `ValueError` type is triggered.

`set_timer(index_or_key, seconds)`

Sets the specified timer to the transferred value. May only be called in the context of the node.

Parameters:

- index\_or\_key - Index or key of the input. If the index or key does not exist, an exception is triggered.
- seconds - Value in seconds after which the timer is triggered. If the value None or 0 is transferred, the timer is stopped. If an invalid value is transferred, an exception is triggered.

Exceptions:

- If the method is not called from the thread of the context, an exception of type Hsl3ContextError is triggered.

Example:

```
# The first timer is called in 120 seconds
self.fw.set_timer(1, 120)
# The second timer is called in 240 seconds
self.fw.set_timer(2, 240)
# The second timer is stopped
self.fw.set_timer(2, 0)
```

`run_in_context(callback, params)`

Calls a method in the context of the node. This method is always necessary if data from an external thread is to be called and processed in the context of the node.

Parameters:

- callback - Method called in the context of the node.
- params - Tuple transferred to the method called as a parameter.

Exceptions:

- If no method is transferred for the callback parameter, an exception of the ValueError type is triggered.
- If no tuple is transferred for the params parameter, an exception of the ValueError type is triggered.

Example:

```
# Calls the method calc_and_send for own instance.
# The three parameters 1, 2 and 3 are transferred to the call
# def calc_and_send(self, p1, p2, p3):
#     pass
self.fw.run_in_context(self.calc_and_send, (1, 2, 3))
```

`get_logger(host, port, console, level)`

Generates a logger with which log messages can be sent via Syslog. The HS-specific Syslog server can be reached at the address 127.0.0.1:65002.

Parameters:

- host - Optional, the destination address of the Syslog server (default: 127.0.0.1)
- port - Optional, the IP port of the Syslog server (default: 65002)
- console - Optional, if True, the outputs are also output on the screen. Should only be used for test purposes (default: False).
- level - Optional, sets the log level (default: logging.INFO). The constants on the logging module are used here.

Example:

```
# No parameter
self.logger = self.fw.get_logger()
# External Syslog server
self.logger = self.fw.get_logger(host="192.168.0.12", port=514)
```

`create_debug_section()`

Creates an instance of the class Hsl3DebugSection.

Example:

```
self.debug = self.fw.create_debug_section()
self.debug.set("Field 1", "?")
self.debug.set("Field 2", "?")
```

`get_instance(instance_id)`

Returns the instance of an HSL3 node.

Parameters:

- instance\_id - ID of an HSL3 node. The instance must belong to the same context.

`get_context_id()`

Returns the ID of the current context (string).

`get_instance_id()`

Returns the ID of the current instance (Int).

`get_module_id()`

Returns the ID of the current module (Int).

## Hsl3DebugSection

An instance of this class is returned when creating a debug section.

The order within the section is determined by the order of the definition.

### Methods

`set(name, value)`

Defines a field in the section and sets its value.

Parameters:

- name - Field name.
- value - Field value.

`inc(name, value=1)`

The field value is increased.

Parameters:

- name - Field name. If the field does not yet exist, the value is initialised with value.
- value - Value by which the content of the field (default: 1) is increased. If the value is not a numerical value, an exception is triggered.

`avg(name, value=None)`

Calculates the average of a value.

Parameters:

- name - Field name. If the field does not yet exist, the value is initialised with value.
- value - If the value is a numerical value, the average is calculated. If None is transferred, the field is reset.

`timestamp(name, value=None)`

Parameters

- name - Field name. If the field does not yet exist, the value is initialised with value.
- value - If set, this value is accepted, otherwise current time stamp.

`log(msg)`

Parameters

- msg - Enters the text in the log.

`exception(msg)`

Parameters

- msg - Enters the current exception in the log and adds the text as additional information.

## Hsl3Slots

Instances of this class are transferred when the following methods are called:

- `on_init(inputs: Hsl3Slots, store: Hsl3Slots)`
  - `inputs` - Contains all inputs
  - `store` - Contains all memory variables
- `on_calc(inputs: Hsl3Slots)`
  - `inputs` - Contains all inputs
- `on_timer(timer: Hsl3Slots)`
  - `timer` - Contains all defined timers

## Access

An instance of the Hsl3Slots class can be accessed directly via the index or key of the individual slot.

Example:

```
def on_init(self, inputs, store):
    # Returns instance of Hsl3Slot
    slot_e1 = inputs[1]
    # Returns the value of Input 1
    slot_e1_wert = inputs[1].value

def on_calc(self, inputs):
    # Retrieves the value from Input 1
    if inputs[1].changed:
        slot_e1_wert = inputs[1].value
```

## Methods

`get(index_or_key)`

Parameters

- `index_or_key`

Return:

Returns an object of type Hsl3Slot or triggers an exception if the index or key does not exist.

`keys()`

Return:

Returns a list of all existing keys.

`changed(index_or_key)`

Parameters

- `index_or_key`

Return:

Returns *True* when a value is received. If no value was received at this input or no value was found for the transferred key, *False* is returned

`value(index_or_key)`

Parameters

- `index_or_key`

Return:

Returns the value. If the corresponding index or key is not found, *None* is returned.

## Hsl3Slot

Instances of this class are transferred when the following methods are called:

### Attribute

#### value

Returns the value of the respective slot:

- Input - The value reflects the value of the input.
- Store - The value reflects the value of the memory variable.
- Timer - The value reflects the duration of the timer.

The value is always of type float, int or bytes.

#### changed

Returns *True* if the value has changed:

- Input - True if the value has changed and a calculation has been triggered (on\_calc call). Always False during initialisation (on\_init call).
- Store - Always False.
- Timer - True if the corresponding timer was triggered (on\_timer call)

## Libraries

The firmware contains all Python 3 standard libraries.

### External libraries

In addition to the standard libraries, the following external libraries are also available:

- requests
- websockets
- beautifulsoup4
- pytz
- python-dateutil
- pymodbus

## HSL3 generator

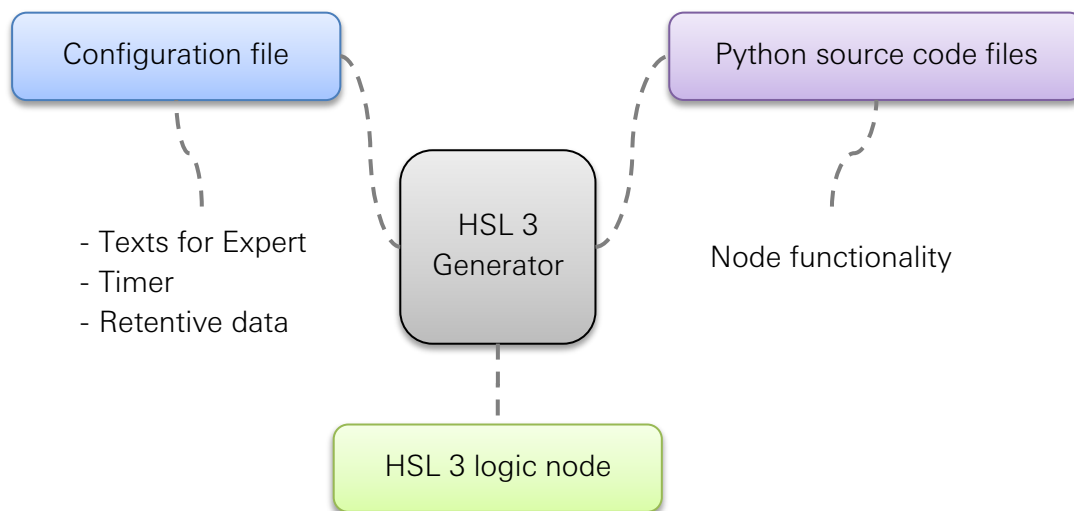
### Overview

The generator is available to facilitate the development of logic nodes.

Using a configuration file and the source code files for the logic node, the generator creates the finished HSL 3 logic node.

Some properties of the logic node and the texts for display in Expert are specified in the configuration file.

### Use of the generator



### Conditions

Python 3x is required to use the generator.

### Generator call up

The generator is called via the command line with the Python interpreter. The following command line parameters are available: **--source** is used to specify the configuration file (default value: **config.xml**). **--target** specifies the target file, which overwrites the configuration file specifications. If this information is not available, the configuration file determines the name of the target file.

Optionally, **--debug** enables debug outputs that are disabled by default.

### Example

```
python generator3x.pyc --source my_config.json --target
10001_my_module.hsl --debug
```

The generator is called using `python generator3x.pyc`. `--source my_config.json` specifies that `my_config.json` should be used as the configuration file. `--target`



`10001_my_module.hsl` specifies that the output file should be called `10001_my_module.hsl`, which overwrites any corresponding specification in the configuration file. `--debug` enables additional debug outputs during generation. As a result of this call, the generator generates the logic node `10001_my_module.hsl`.

Description of command line parameters

Parameter s	Short form	Description	Example value	Default value
<code>--source</code>	<code>-s</code>	Configuration file	<code>my_config.json</code>	<code>config.xml</code>
<code>--target</code>	<code>-t</code>	Target file (overwrites specification of the configuration file)	<code>10001_my_module.hsl</code>	Configuration file specifies the name
<code>--debug</code>	<code>-d</code>	Enables debug outputs		Debug outputs disabled

## Configuration file

The configuration file describes the basic properties of the logic node, such as name, ID, category, number, designations and types of inputs and outputs etc. The configuration file can be in JSON or XML file formats. Both offer the same range of parameters or attributes for creating the HSL3 logic node.

Attributes of the configuration file

### **module**

Defines a single logic node.

#### **category**

Node category for the GLE (graphical logic editor) in Expert.

#### **context**

Context in which the node is defined. See also: Threading.

#### **id**

Logic node ID

#### **name**

Name in the GLE (graphical logic editor in Expert)

#### **version**

Version name of the node

#### **inputs**

Logic node inputs

#### **outputs**

Logic node outputs

#### **stores**

Logic node retentive variables

#### **timers**

Logic node timers

## **scripts**

Python source code files of the logic node (path specifications relative to the configuration file)

## **translations**

Contains all the logic node texts visible in the GLE. One translation must be specified per language.

If the texts are not available in the set language, Expert uses the texts defined for the module, input and output tags.

## **input**

Defines a logic node input.

### **type**

Input type: *number*, *string*, *base\_path* or *destination\_port*

number: Numerical value

string: Alphanumeric value/character string

base\_path: Port at which the node must listen if it is to be reached from outside via the base path.

destination\_port: Base path via which all calls are forwarded to the node. Is attached to the HomeServer URL.

### **init\_value**

Value with which the input is initialised.

### **label**

Designation of the input as it is displayed in the GLE.

### **identifier**

Designation of the input as used in the Python source code.

## **output**

Defines a logic node output.

### **type**

Output type: *number* or *string*

number: Numerical value

string: Alphanumeric value/character string

### **init\_value**

Value with which the output is initialised.

### **label**

Designation of the output as it is displayed in the GLE.

### **identifier**

Designation of the output as used in the Python source code.

## **store**

Defines a retentive variable of the logic node.

### **type**

Type: *number* or *string*

number: Numerical value

string: Alphanumeric value/character string

**init\_value**

Value with which the retentive variable is initialised.

**identifier**

Designation of the retentive variable as used in the Python source code.

**timer**

Defines a timer used by the logic node.

**identifier**

Designation of the timer as used in the Python source code.

**script**

Specification of the Python source code files used by the logic node.

**filename**

Python source code file (path details relative to the configuration file).

The file name must have the prefix `hs13_<ID>`.

e.g. `hs13_10001_my_module_src.py`

**folder**

Directory with Python source code files (path details relative to the configuration file).

Account is taken of all files in the directory that have the prefix `hs13_<ID>`.

**translation**

Defines the node text for a language.

**language**

Language abbreviation. All languages supported by Expert are valid.

**name**

Name in the GLE (graphical logic editor in Expert) in *language*.

**category**

Node category for the GLE (graphical logic editor) in Expert in *language*.

**translation\_inputs**

Contains the names for all node inputs in *language*.

**translation\_outputs**

Contains the names for all node outputs in *language*.

**translation\_input**

Name for an input in *language*.

**label**

Designation of the input in *language* as it is displayed in the GLE.

**translation\_output**

Name for an output in *language*.

**label**

Designation of the output in *language* as it is displayed in the GLE.

## HSL 3 - Communication between logic node instances

### Methods in the framework

The HSL 3 Framework offers the option of enabling logic nodes to access instances of other nodes.

This enables communication between logic node instances.

There is a method in the framework for obtaining an instance of a logic node:

**`get_instance(instance_id)`**

As a result, the methods return the logic node instance associated with `instance_id` (or "None" if the `instance_id` is invalid).

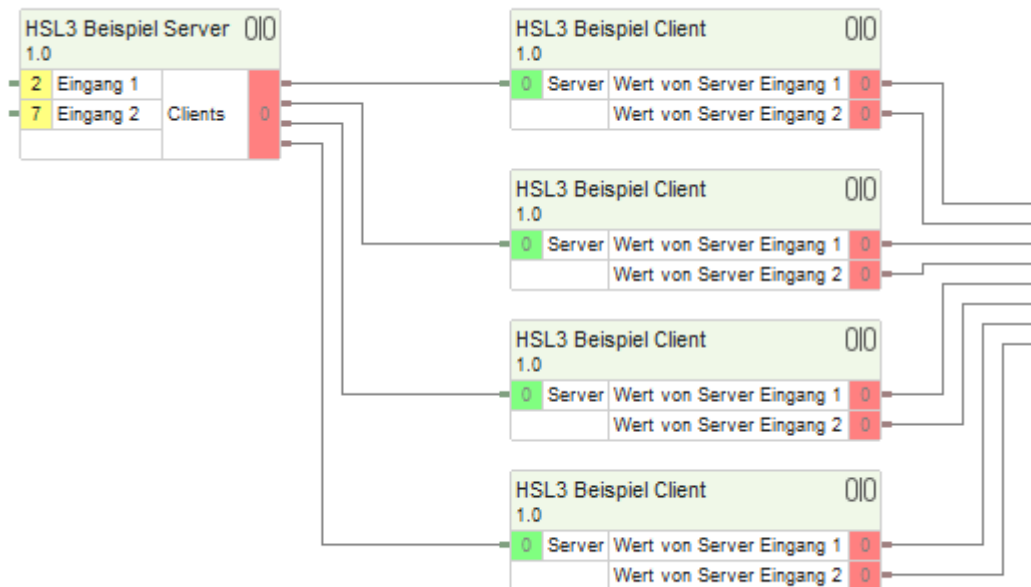
The `instance_id` is an ID uniquely assigned by the framework during the runtime of the HomeServer for the respective instance of a node. It does not match the five-digit node ID (e.g. 19042). The `instance_id` (also called runtime ID) can be determined using the `get_instance_id()` method. The following example code would return its own instance (i.e. `self`):

```
self.fw.get_instance(self.fw.get_instance_id())
```

The runtime ID can be easily exchanged and made known by wiring inputs and outputs between logic nodes.

## Example

The following example shows the communication between logic node instances:



The source code for the logic nodes, in this example, is enclosed with the package.

## Process of communication

1. The logic node "HSL3 Example Server" queries its instance\_id during initialisation with the method `self.fw.get_instance_id` and writes it to its output.
2. The logic nodes "HSL3 Example Client" receive the instance\_id to their inputs and receive the instance of the logic node "HSL3 Example Server" with the method `self.fw.get_instance`.
3. Triggered by a timer, the logic nodes "HSL3 Example Client" query the values of the inputs of the logic node "HSL3 Example Server" by accessing its instance directly. They then write the queried values to their outputs.

## Notes on migration from HSL2 to HSL3

### Introduction

The introduction of the HSL 2 Framework SDK extended the functionality of the Gira HomeServer and Gira FacilityServer from firmware version 4.5 onwards to include the option of using the full Python language scope in the HS logic nodes.

Python versions 2.6 and 2.7 were supported for this purpose.

As of firmware version 4.13, it is now possible, with the aid of HSL3 SDK, for users to develop their own logic modules for the Gira HomeServer or Gira FacilityServer, which use Python 3.9.

The HSL3 SDK therefore replaces the HSL 2 SDK in the long term and it is therefore necessary to migrate logic nodes implemented with HSL 2 to HSL 3.

Below is an explanation of what needs to be taken into account.

### Differences between HSL 2 and HSL 3

In addition to the use of Python 3 instead of Python 2, as mentioned above, there are several other differences between HSL 2 and HSL 3:

<b>HSL2</b>	<b>HSL3</b>
Uses Python 2.	Uses Python 3.
HSL 2 Framework is part of the logic node.	HSL 3 Framework is part of the firmware.
The framework includes support for encryption and network communication.	The scope of the framework is reduced to the essentials.
Specified project structure.	Project structure can be individually designed.
Configuration file for the generator in XML.	Configuration file for the generator in JSON or XML.
The logic node source code must consist of one file.	The logic node source code can consist of several files.
It is possible to integrate own Python modules via import, but this is restricted to generator-based imports.	Integration of Python modules currently not planned.

### Tips for migrating logic nodes from HSL 2 to HSL 3

#### Converting Python 2 to Python 3 source code with 2to3

Migrating source code from Python 2 to Python 3 is not a trivial process, as the syntax, standard library and behaviour have changed in some core areas.

2to3 is an official tool from the Python standard package (included up to Python version 3.12.10), which can translate the source code from Python 2 to Python 3.

It works on the basis of predefined rules that identify and adapt typical differences between the two language versions.

The tool is suitable for conversion to Python 3 because it is systematic and reproducible. However, it does not cover all cases: Manual reworking is often required, especially when handling strings and byte sequences. For successful use, it is recommended that the converted code be thoroughly tested and the automatic adjustments critically checked.

## Strings and bytes

In Python 2, the *str* type is a sequence of bytes. It does not contain any information about character encoding, but merely represents a byte sequence. If you really want to display text in the sense of Unicode characters, you have to use the *unicode* type. Such objects can be created with the prefix *u*. Conversions between *str* and *unicode* are always explicitly made via *encode()* and *decode()*.

This means that in Python 2, *str* stands for bytes, while *unicode* stands for real text.

With Python 3, this model has been designed more logically. Here, *str* is always a sequence of Unicode characters, i.e. text. If you need a byte sequence, you can use the *bytes* type, which is written with a prefix *b*. Direct mixing of *str* and *bytes* is not permitted. To convert text to bytes, you must explicitly call *encode()*, and to convert bytes back to text, you must call *decode()*.

This must be borne in mind when using inputs, outputs and memory variables of logic nodes in HSL 3. The strings that are forwarded by the logic to the logic node or which the logic node outputs to the logic must be of *bytes* type.

Example of writing a text of type *str* to an output:

```
self.fw.set_output("Ausgang Text",  
mein_text_str.encode(encoding="ascii",errors="ignore"))
```

## Use of retentive memory variables

If the content of a retentive memory variable is to be available to the logic node after migration, it must be ensured that two dummy variables are created in HSL 3.

Example for HSL 2:

```
<remanent_variables>
  <remanent_variable
const_name="slot_1">slot_1</remanent_variable>
</remanent_variables>
```

Example for HSL 3:

```
<stores>
  <store type="string" identifier="dummy1" init_value="" />
  <store type="string" identifier="dummy2" init_value="" />
  <store type="string" identifier="slot_1" init_value="" />
</stores>
```

The memory variables *dummy1* and *dummy2* are only placeholders and have no further meaning for the logic node.

If a new HSL3 node is created, it is not necessary to create the two *dummies*. This is only used for compatibility with the retentive data of the HSL2 predecessor nodes.