Open for Innovation
KNIME

Create a New Python based KNIME Extension

KNIME AG, Zurich, Switzerland
Version 4.7 (last updated on 2023-03-27)

Table of Contents

Introduction

As explained in the Extensions and Integrations Guide, KNIME Analytics Platform can be enhanced with additional functionality provided by a vast array of extensions and integrations. Often, installing an extension adds a collection of new nodes to the node repository of KNIME Analytics Platform.

With the v4.6 release of KNIME Analytics Platform, we introduce the possibility to write KNIME node extensions completely in Python. This includes the ability to define node configuration and execution, as well as dialog definition. A Pythonic API to design those nodes is now available, as well as debugging functionality within KNIME Analytics Platform. This means deploying pure-Python KNIME extensions containing nodes – including their Python environment needed for execution – using a locally built update site is now possible.

In this guide, we offer a tutorial to get you started with writing your KNIME nodes using Python, as well as how to setup a shareable Python extension containing your nodes, together with a complete definition of the API.

Quickstart Tutorial

This section provides a basic extension template, and walks you through the essential development steps to help you get started with using the API.

Prerequisites

1. Set up `conda`.

   To get started with developing Python node extensions, you need to have `conda` installed. Here is the quickest way:

   - Go to the Miniconda website
   - Download the appropriate installer for your OS
   - For Windows and macOS: run the installer executable
   - For Linux: execute the script in terminal (see here for help)

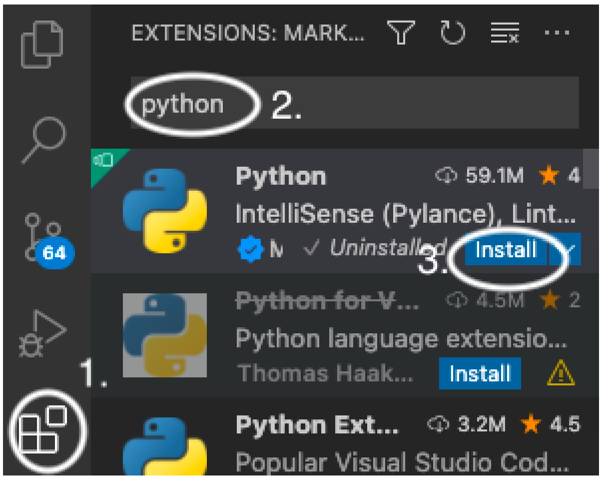
2. With `conda` set up, extract basic.zip to a convenient location.

   In the `basic` folder, you should see the following file structure:

```
.
├──── tutorial_extension
│     │──── icon.png
│     │──── knime.yml
│     │──── LICENSE.TXT
│     └──── my_extension.py
├──── config.yml
├──── my_conda_env.yml
├──── Example_with_Python_node.knwf
└──── README.md
```

3. During development, you can edit the source files in any text editor. However, in order to make use of autocompletion for the API, as well as to allow debugging via the `debugpy` package, we recommend using an editor that is able to set `conda` environments as the Python interpreter. Here are the setup steps for **Visual Studio Code**:

   - Download and install Visual Studio Code
   - Install the Python extension



- In the bottom right corner of the editor, you should be able to select the Python interpreter that you would like to use during development. After Step 4 of Tutorial 1, you will have the `my_python_env` environment available in the list of Python interpreters. By selecting the environment, you will be able to make full use of autocompletion.

Writing your first Python node from scratch

This is a quickstart guide that will walk you through the essential steps of writing and running your first Python node extension containing a single node. We will use `tutorial_extension` as the basis. The steps of the tutorial requiring modification of the Python code in `my_extension.py` have corresponding comments in the file, for convenience.

For an extensive overview of the full API, please refer to the Defining a KNIME Node in Python: Full API section, as well as our Read the Docs page.

1. Install KNIME Analytics Platform version 4.6.0 or higher.
2. Go to *File* → *Install KNIME Extensions…*, enter "Python" in the search field, and look for *KNIME Python Extension Development (Labs)*. Alternatively, you can manually navigate to the *KNIME Labs Extensions* category and find the extension there. Select it and proceed with installation.
3. The `tutorial_extension` will be your new extension. Familiarize yourself with the files contained in that folder, in particular:
    - `knime.yml`, which contains important metadata about your extension.
    - `my_extension.py`, which contains Python definitions of the nodes of your extension.

- `config.yml`, just outside of the folder, which contains the information that binds your extension and the corresponding `conda`/Python environment with KNIME Analytics Platform.

4. Create a `conda`/Python environment containing the `knime-python-base` metapackage, together with the node development API `knime-extension`. If you are using `conda`, you can create the environment by running the following command in your terminal (macOS/Linux) or Anaconda Prompt (Windows):

```
conda create -n my_python_env python=3.9 knime-python-base=4.7 knime-extension=4.7
-c knime -c conda-forge
```

If you would like to install the packages into an environment that already exists you can run the following command *from within that environment*:

```
conda install knime-python-base=4.7 knime-extension=4.7 -c knime -c conda-forge
```

Note that you **must** append both the `knime` and `conda-forge` channels to the commands to install the mandatory packages. To install additional packages, for your specific use case, we recommend using the `conda-forge` channel.

```
conda install -c conda-forge <additional_pkg_name>
```

5. Edit the `config.yml` file located just outside of the `tutorial_extension` (for this example, the file already exists with prefilled fields and values, but you would need to manually create it for future extensions that you develop). The contents should be as follows:

```
<extension_id>:
    src: <path/to/folder/of/template>
    conda_env_path: <path/to/my_python_env>
    debug_mode: true
```

where:

- `<extension_id>` should be replaced with the `group_id` and `name` values specified in `knime.yml`, combined with a dot.

  For our example extension, the value for `group_id` is `org.tutorial`, and the value for `name` is `first_extension`, therefore the `<extension_id>` placeholder should be replaced with `org.tutorial.first_extension`.

- The `src` field should specify the path to the `tutorial_extension` folder.

  For instance,
  `/Users/Bobby/Development/python_extensions/tutorial_extension`

- Similarly, the `conda_env_path` field should specify the path to the `conda`/Python environment created in Step 4. To get this path, run the `conda env list` command in your Terminal/Anaconda Prompt, and copy the path displayed next to the appropriate environment (`my_python_env` in our case).

- The `debug_mode` is an optional field, which, if set to `true`, will tell KNIME Analytics Platform to use the latest changes in the `configure` and `execute` methods of your Python node class whenever those methods are called.

  > Enabling `debug_mode` will affect the responsiveness of your nodes.

6. We need to let KNIME Analytics Platform know where the `config.yml` is in order to allow it to use our extension and its Python environment. To do this, you need to edit the `knime.ini` of your KNIME Analytics Platform installation, which is located at `<path-to-your-KAP>/knime.ini`.

   Append the following line to the end, and modify it to have the correct path to `config.yml`:

   ```
   -Dknime.python.extension.config=<path/to/your/config.yml>
   ```

   > The forward slash `/` has to be used on all OS, also on Windows.

7. Start your KNIME Analytics Platform.

8. The "My Template Node" node should now be visible in the Node Repository.

9. Import and open the `Example_with_Python_node.knwf` workflow, which contains our test node:

   a. Get familiar with the table.

   b. Study the code in `my_extension.py` and compare it with the node you see in KNIME Analytics Platform. In particular, understand where the node name and description, as well as its inputs and outputs, come from.

   c. Execute the node and make sure that it produces an output table.

10. Build your first configuration dialog:

    In `my_extension.py`, uncomment the definitions of parameters (marked by the "Tutorial Step 10" comment). Restart your KNIME Analytics Platform, re-drag your node from the node repository into the workflow, and you should be able to double-click the node and see configurable parameter.

Take a minute to see how the names, descriptions, and default values compare between their definitions in `my_extension.py` and the node dialog.

11. Add your first port:

    To add a second input table to the node, follow these steps (marked by the "Tutorial Step 11" comment; you will need to restart KNIME Analytics Platform):

    a. Uncomment the `@knext.input_table` decorator.

    b. Change the `configure` method’s definition to reflect the changes in the schema.

    c. Change the `execute` method to reflect the addition of the second input table.

12. Add some functionality to the node:

    With the following steps, we will append a new column to the first table and output the new table (the lines requiring to be changed are marked by the "Tutorial Step 12" comment):

    a. To inform downstream nodes of the changed schema, we need to change it in the return statement of the `configure` method; for this, we append metadata about a column to the output schema.

    b. Everything else is done in the `execute` method:

        - we transform both input tables to pandas dataframes and append a new column to the first dataframe
        - we transform that dataframe back to a KNIME table and return it

13. Use your parameters:

    a. In the `execute` method, uncomment the lines marked by the "Tutorial Step 13" comment.

    b. Use a parameter to change some table content; we will use a lambda function for a row-wise multiplication using the `double_param` parameter.

14. Start logging and setting warnings:

    Uncomment the lines marked by "Tutorial Step 14" in the `execute` method:

    a. Use the `LOGGER` functionality to inform users, or for debugging.

    b. Use the `execute_context.set_warning("A warning")` to inform users about unusual behaviour.

c. If you want the node to fail, you can raise an exception. For instance: `raise ValueError("This node failed just because")`.

15. Congratulations, you have built your first functioning node entirely in Python!

Python Node Extension Setup

A Python node extension needs to contain a YAML file called `knime.yml` that gives general information about the node extension, which Python module to load, and what conda environment should be bundled with the extension.

knime.yml:

```
name: myextension # Will be concatenated with the group_id to form the extension ID
author: Jane Doe
env_yml_path: <path/to/my_conda_env.yml> # Path to the Conda environment yml, from which
the environment for this extension will be built when bundling
extension_module: my_extension # The .py Python module containing the nodes of your
extension
description: My New Extension # Human readable bundle name / description
long_description: This extension provides functionality that everyone wants to have. #
Text describing the extension (optional)
group_id: org.knime.python3.nodes # Will be concatenated with the name to form the
extension ID
version: 0.1.0 # Version of this Python node extension. Must use three-component
semantic versioning for deployment to work.
vendor: KNIME AG, Zurich, Switzerland # Who offers the extension
license_file: LICENSE.TXT # Best practice: put your LICENSE.TXT next to the knime.yml;
otherwise you would need to change to path/to/LICENSE.txt
#Optional: If you do not have dependencies on other extensions, you do not need
feature_depencendies and their entries
feature_dependencies:
  - org.knime.features.chem.types 4.7.0 # If you want to specify the version
  - org.knime.features.chem.types       # If the version does not matter
```

The `id` of the extension will be of the form `group_id.name`. It needs to be a unique identifier for your extension, so it is a good idea to encode your username or company's URL followed by a logical structure as `group_id` in order to prevent `id` clashes. For example, a developer from KNIME could encode its URL to `org.knime` and add `python3` to indicate that the extension is a member of `nodes`, which are part of `python3`.

Feature dependencies: if your extension depends on another extension, you can specify it as a bullet point of `feature_dependencies`. Optionally, you can add a specific version to it.

Example: You use data types like `SmilesValue` of the `KNIME Base Chemistry Types & Nodes` extension in your extension. You have that extension already installed and want to make sure that everybody who uses your extension will also have this extension installed. Then you can go to *Help > About KNIME Analytics Platform > Installation Details* and check the id of `KNIME Base Chemistry Types & Nodes`, which is `org.knime.features.chem.types.feature.group`. Take the id without `.feature.group` and you have the string of the feature dependency:


```
org.knime.features.chem.types
```

Note that the `env_yml_path` field, which specified the path to the YAML configuration of the `conda` environment required by your extension, is needed when bundling your extension. During development, KNIME Analytics Platform uses the environment specified in the `config.yml` file.

The path containing the `knime.yml` will then be put on the `Pythonpath`, and the extension module specified in the YAML will be imported by KNIME Analytics Platform using `import <extension_module>`. This Python module should contain the definitions of KNIME nodes. Each class decorated with `@knext.node` within this file will become available in KNIME Analytics Platform as a dedicated node.

Recommended project folder structure:

```
.
├─── my_extension
│     ├─── icons
│     │     └─── my_node_icon.png
│     ├─── knime.yml
│     ├─── LICENSE.txt
│     ├─── my_conda_env.yml
│     └─── my_extension.py
└─── config.yml
```

See Tutorial 1 above for an example.

Development and distribution

As you develop your Python extension, you are able to run and debug it locally by setting the `knime.python.extension.config` system property in your KNIME Analytics Platform's `knime.ini` to point to the `config.yml`, or in the launch configuration's VM arguments in Eclipse. See the Registering Python extensions during development and Customizing the Python executable sections at the end of this guide for more information.

In order to share your Python extension with others, please refer to the Bundling your Python Extension Nodes section.

Defining a KNIME Node in Python: Full API

We provide a `conda` package that includes the full API for node development in Python - `knime-extension` (see Tutorial 1 for help in setting up your development `conda` environment). To enable helpful code autocompletion via `import knime.extension as knext`, make sure your IDE of choice's Python interpreter is configured to work in that `conda` environment when you are developing your Python node extension (see here for help with Visual Studio Code, here for PyCharm, here for Sublime Text, or here for general information on integrating your IDE with `conda`).

A Python KNIME node needs to implement the `configure` and `execute` methods, so it will generally be a class. The node description is *automatically generated from the docstrings* of the class and the `execute` method. The node's location in KNIME Analytics Platform's *Node Repository*, as well as its icon, are specified in the `@knext.node` decorator.

A simple example of a node does nothing but pass an input table to its output unmodified. Below, we define a class `MyNode` and indicate that it is a KNIME node by decorating it with `@knext.node`. We then "attach" an input table and an output table to the node by decorating it with `@knext.input_table` and `@knext.output_table` respectively. Finally, we implement the two required methods, `configure` and `execute`, which simply return their inputs unchanged.

```
import knime.extension as knext

@knext.node(name="My Node", node_type=knext.NodeType.MANIPULATOR,
icon_path="../icons/icon.png", category="/")
@knext.input_table(name="Input Data", description="The data to process in my node")
@knext.output_table("Output Data", "Result of processing in my node")
class MyNode:
    """Short description is in the first line next to the three double quotes here. It
it displayed in overviews when a whole category in the node repository is selected.

    Here begins the normal description: This node description will be displayed in KNIME
Analytics Platform.
    """
    def configure(self, config_context, input_table_schema):
        return input_table_schema

    def execute(self, exec_context, input_table):
        return input_table
```

`@knext.node`'s configuration options are:

- **name**: the name of the node in KNIME Analytics Platform.

- **node_type**: the type of the node, one of:
  - `knext.NodeType.MANIPULATOR`: a node that manipulates data.
  - `knext.NodeType.LEARNER`: a node learning a model that is typically consumed by a PREDICTOR.
  - `knext.NodeType.PREDICTOR`: a node that predicts something typically using a model provided by a LEARNER.
  - `knext.NodeType.SOURCE`: a node producing data.
  - `knext.NodeType.SINK`: a node consuming data.
  - `knext.NodeType.VISUALIZER`: a node that visualizes data.
  - `knext.NodeType.OTHER`: a node that doesn't fit any of the other node types.
- **icon_path**: module-relative path to a 16x16 pixel PNG file to use as icon.
- **category**: defines the path to the node inside KNIME Analytics Platform's *Node Repository*.

Defining custom port objects

Besides tables, a node can also consume or produce other port objects and it is possible to define custom port objects for your extension. You can do so by extending `knext.PortObject` and `knext.PortObjectSpec` with your custom implementation. In order to use these objects in your node, you need to define a custom port type via the `knext.port_type` function that takes your `PortObject` and `PortObjectSpec` classes as well as a human-readable name for your port type and an optional id. Here is an example:

Let's start with the `PortObjectSpec`:


```
import knime.extension as knext

class MyPortObjectSpec(knext.PortObjectSpec):
    def __init__(self, spec_data: str) -> None:
        self._spec_data = spec_data

    def serialize(self) -> dict:
        return {"spec_data": self._spec_data}

    @classmethod
    def deserialize(cls, data: dict) -> "MyPortObjectSpec":
        cls(data["spec_data"])

    @property
    def spec_data(self) -> str:
        return self._data
```

The `serialize` and `deserialize` methods are used by the framework to store and load the spec.

Note: The `deserialize` method must be a `classmethod`.

The `spec_data` property is just an example for custom code and you can add arbitrary methods to your spec class as you see fit.

Next we implement the `PortObject`:

```
import pickle

class MyPortObject(knext.PortObject):
    def __init__(self, spec: MyPortObjectSpec, model) -> None:
        super().__init__(self, spec)
        self._model = model

    def serialize(self) -> bytes:
        return pickle.dumps(self._model)

    @classmethod
    def deserialize(cls, spec: MyPortObjectSpec, data: bytes) -> "MyPortObject":
        return cls(spec, pickle.loads(data))

    def predict(self, data):
        return self._model.predict(data)
```

The PortObject class must have a `serialize` and `deserialize` method that are called by the framework to persist and restore the object. Again note that `deserialize` has to be a `classmethod`.

The `predict` property is again just an example for custom code that your port object class may contain.

Finally, we create a custom port type to be used as input or output of a node:

```
my_model_port_type = knext.port_type(name="My model port type",
object_class=MyPortObject, spec_class=MyPortObjectSpec)
```

The `knext.port_type` method ties the `PortObject` and `PortObjectSpec` together and provides a human-readable name to refer to the custom port type.

It is also possible to specify a custom ID for the port type via the `id` argument. Note that the id must be unique! If you don't provide a custom ID, then the framework generates one of the format `your_extension_id.your_module_name.your_port_object_class_name`. For example if your extension has the id `org.company.extension` and you implement a `MyPortObject` in the module `my_extension`, then the generated id is `org.company.extension.my_extension.MyPortObject`.

Check out the next section to learn how to declare your custom port type as input or output of your node.

Node port configuration

The input and output ports of a node can be configured by decorating the node class with `@knext.input_table`, `@knext.input_port`, and respectively `@knext.output_table` and `@knext.output_port`. Additionally, an output port producing a view can be added with the `@knext.output_view` decorator.

These port decorators have the following properties:

- they take `name` and `description` arguments, which will be displayed in the node description area inside KNIME Analytics Platform;
- they must be positioned *after* the `@knext.node` decorator and *before* the decorated object (e.g. the node class);
- their order determines the order of the port connectors of the node in KNIME Analytics Platform.

The `@knext.input_table` and `@knext.output_table` decorators configure the port to consume and respectively produce a KNIME table.

If you want to receive or send other data, e.g. a trained machine learning model, use

`@knext.input_port` and `@knext.output_port`. These decorators *have an additional argument*, `port_type`, used to identify the type of port objects going along this port connection. Only ports with equal `port_type` can be connected. See the previous section to learn how to specify your own port type.

The port configuration determines the expected signature of the `configure` and `execute` methods:

- In the `configure` method, the first argument is a `ConfigurationContext`, followed by one argument per input port. The method is expected to return **as many parameters as it has output ports configured**. The argument and return value types corresponding to the input and output ports are:
  - for **table** ports, the argument/return value must be of type `knext.Schema`. If the return table consists of only one column, the return value can also be of type `knext.Column`;
  - for **custom** ports, the argument/return value must be of your custom implementation of `knext.PortObjectSpec`. If we take the example from the previous section, the type would be `MyPortObjectSpec`.

    Note that the order of the arguments and return values must match the order of the input and output port declarations via the decorators.
- The arguments and expected return values of the `execute` method follow the same schema: one argument per input port, one return value per output port.

Examples how to use `knext.Schema` and `knext.Column`` (see the API):

```
def configure(self, config_context): # no input table
  """ This node creates a table with a single column """
  ktype = knext.string()
  # OR
  ktype = knext.int32() # OR knext.double(), knext.bool_(), knext.list_(knext.string()),
knext.struct(knext.int64(), knext.bool_()),...
  # OR
  import datetime
  ktype = datetime.datetime
  return knext.Column(ktype, "Date and Time")
```



```
def configure(self, config_context): # no input table
  """ This node creates two tables with two columns each """
  ktype1 = knext.string()
  import knime.types.chemistry as cet # needs the extension `KNIME Base Chemistry Types
& Nodes` installed
  ktype2 = cet.SdfValue
  schema1 = knext.Schema([ktype1, ktype2], ["Column with Strings", "Column with Sdf"])
  schema2 = knext.Schema([ktype1, ktype2], ["Another column with Strings", "Another
column with Sdf"])
  return schema1, schema2
```

All supported types of your current environment can be obtained by printing `knime.api.schema.supported_value_types()` or `knime.extension.supported_value_types()`` .

Here is an example with two input ports and one output port. See the previous session for the definitions of `MyPortObject`, `MyPortObjectSpec` and `my_model_port_type`.

```
@knext.node("My Predictor", node_type=knext.NodeType.PREDICTOR, icon_path="icon.png",
category="/")
@knext.input_port("Trained Model", "Trained fancy machine learning model",
port_type=my_model_port_type)
@knext.input_table("Data", "The data on which to predict")
@knext.output_table("Output", "Resulting table")
class MyPredictor():
    def configure(self, config_context: knext.ConfigurationContext, input_spec:
MyPortObjectSpec, table_schema: knext.Schema) -> knext.Schema:
        # We will add one column of type double to the table
        return table_schema.append(knext.Column(knext.double(), "Predictions"))
        # If you want to use types known to KNIME, but that have no dedicated KNIME
type, you could use:
        # import datetime
        # return table_schema.append(knext.Column(datetime.datetime, "Date and Time"))

    def execute(self, exec_context: knext.ExecutionContext, trained_model: MyPortObject,
input_table: knext.Table) -> knext.Table:
        predictions = trained_model.predict(input_table.to_pandas())
        output_table = input_table
        output_table["Predictions"] = predictions
        return knext.Table.from_pandas(output_table)
.
```

Alternatively, you can populate the `input_ports` and `output_ports` attributes of your node class (on class or instance level) for more fine grained control.

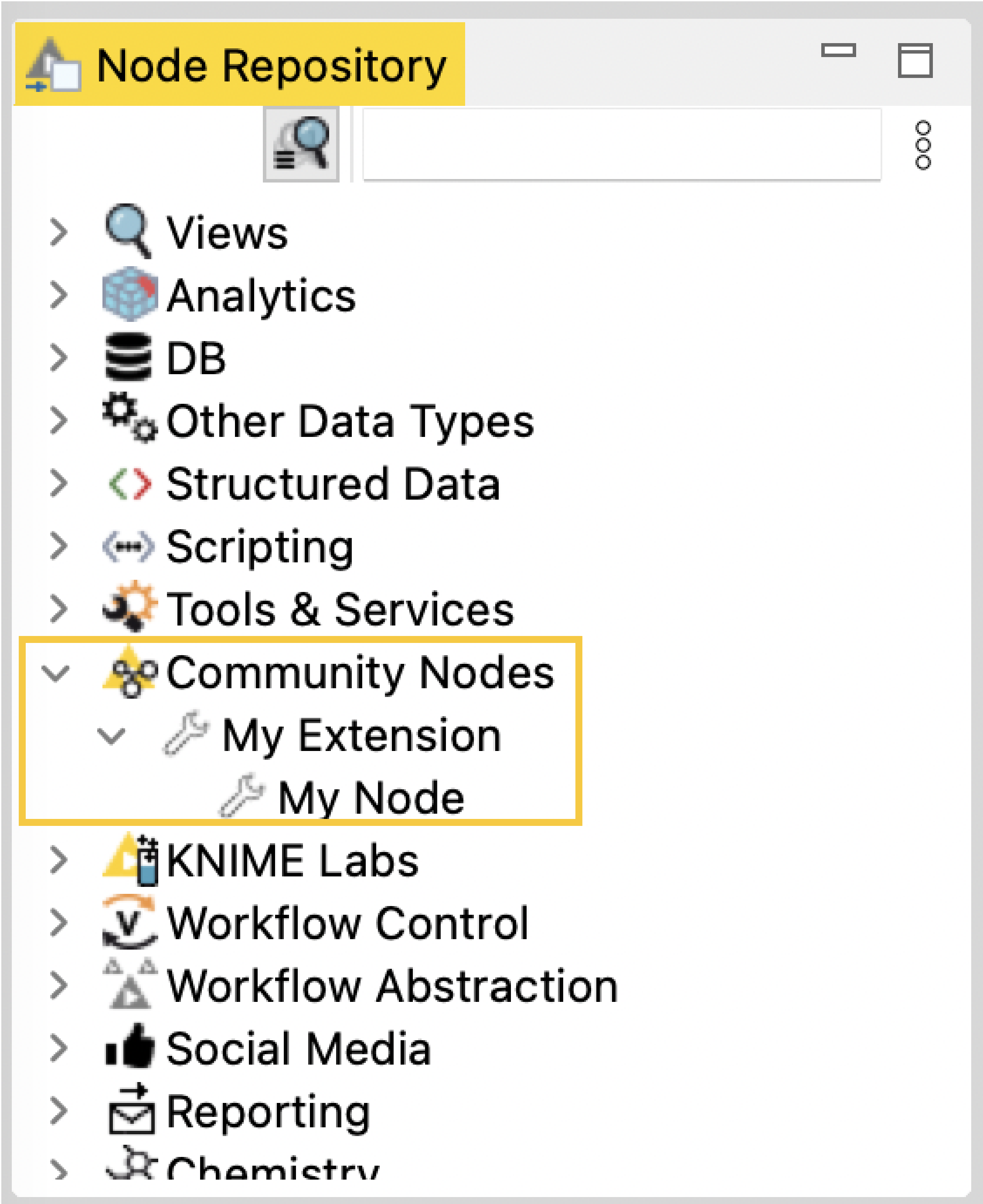
Specifying the node category

Each node in your Python node extension is assigned a category via the `category` parameter of the `@knext.node` decorator, which dictates where the node will be located in the node repository of KNIME Analytics Platform. Without an explicit category, the node will be placed in the root of the node repository, thus you should **always** specify a category for each node.

In order to define a custom category for the nodes of your extension, you can use the `knext.category` helper function. If autocompletion is enabled in your IDE, you should be able to see the list of the expected parameters of the function, together with their detailed description.

If you are a *community developer*, you should use the **Community Nodes** category as the parent category of your Python node extensions. This is done by specifying the `path="/community"` parameter of the `knext.category` function:

```
import knime.extension as knext

my_category = knext.category(
    path="/community",
    level_id="my_extension",
    name="My Extension",
    description="My Python Node Extension.",
    icon="icon.png",
)

@knext.node(
    name="My Node",
    node_type=knext.NodeType.PREDICTOR,
    icon_path="icon.png",
    category=my_category
)
...
class MyNode():
    ...
.
```

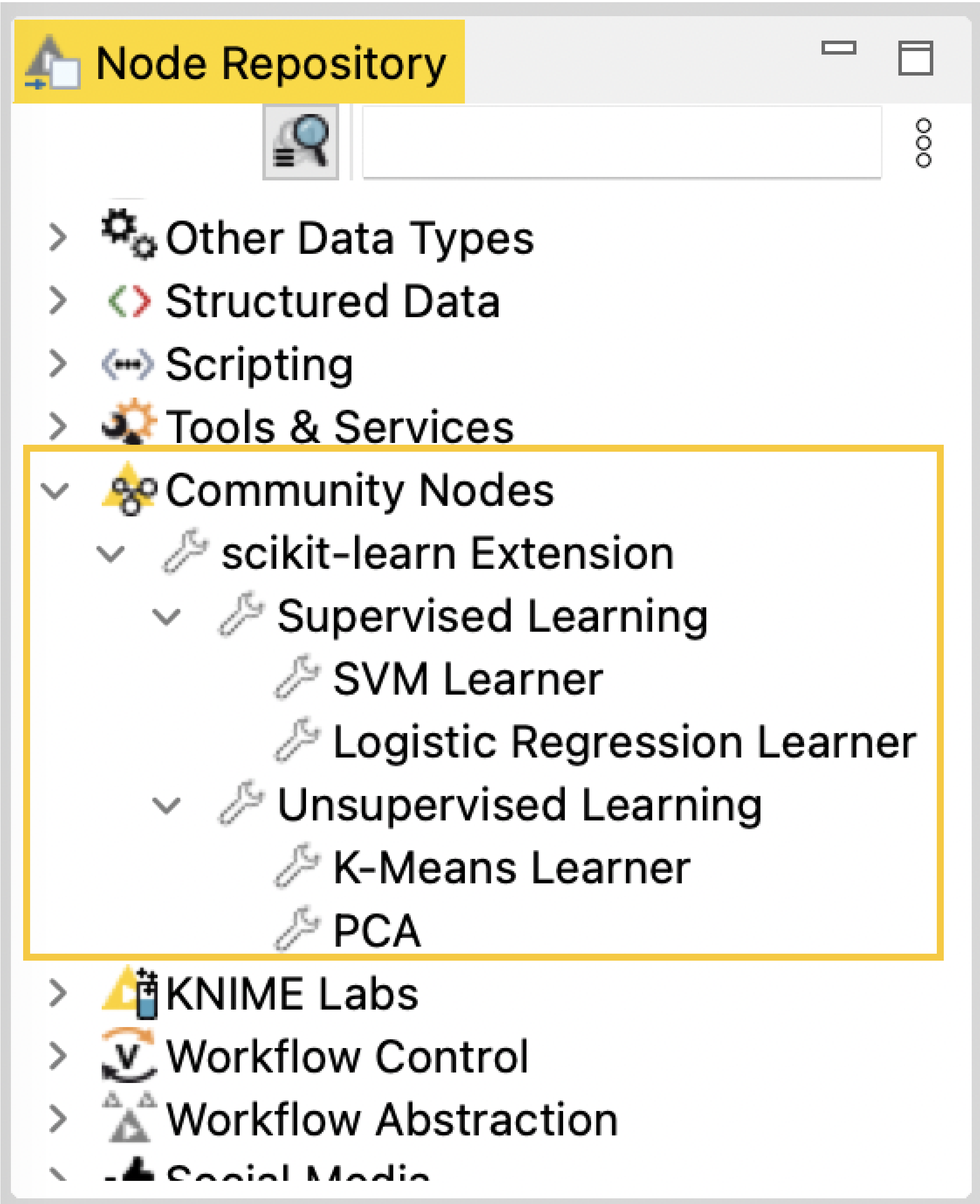
Note that it is possible to further split your custom category into subcategories. This is useful if, for instance, nodes of your extension can be grouped based on their functionality. By first defining a parent category for the node extension, you can then use it as the `path` parameter when defining the subcategories:

```
import knime.extension as knext

# define the category and its subcategories
main_category = knext.category(
    path="/community",
    level_id="my_extension",
    name="scikit-learn Extension",
```



```
    description="Nodes implementing various scikit-learn algorithms.",
    icon="icon.png",
)
supervised_category = knext.category(
    path=main_category,
    level_id="supervised_learning",
    name="Supervised Learning",
    description="Nodes for supervised learning.",
    icon="icon.png",
)
unsupervised_category = knext.category(
    path=main_category,
    level_id="unsupervised_learning",
    name="Unsupervised Learning",
    description="Nodes for unsupervised learning.",
    icon="icon.png",
)

# define nodes of the extension
@knext.node(
    name="Logistic Regression Learner",
    node_type=knext.NodeType.SINK,
    icon_path="icon.png",
    category=supervised_category
)
...
class LogisticRegressionLearner():
    ...


@knext.node(
    name="SVM Learner",
    node_type=knext.NodeType.SINK,
    icon_path="icon.png",
    category=supervised_category
)
...
class SVMLearner():
    ...


@knext.node(
    name="K-Means Learner",
    node_type=knext.NodeType.SINK,
    icon_path="icon.png",
    category=unsupervised_category
)
...
class KMeansLearner():
    ...
```

```
@knext.node(
    name="PCA Learner",
    node_type=knext.NodeType.SINK,
    icon_path="icon.png",
    category=unsupervised_category
)
...
class PCALearner():
    ...
.
```



Defining the node's configuration dialog

> For the sake of brevity, in the following code snippets we omit repetitive portions of the code whose utility has already been established and demonstrated earlier.

In order to add parameterization to your node's functionality, we can define and customize its configuration dialog. The user-configurable parameters that will be displayed there, and whose values can be accessed inside the `execute` method of the node via `self.param_name`, are set up using the following parameter classes available in `knext`:

- `knext.IntParameter` for integer numbers:
  - Signature:

```
knext.IntParameter(
    label=None,
    description=None,
    default_value=0,
    min_value=None,
    max_value=None,
    since_version=None,
)
```

  - Definition within a node/parameter group class:

```
no_steps = knext.IntParameter("Number of steps", "The number of repetition
steps.", 10, max_value=50)
```

  - Usage within the `execute` method of the node class:

```
for i in range(self.no_steps):
    # do something
```

- `knext.DoubleParameter` for floating point numbers:
  - Signature:


```
knext.DoubleParameter(
    label=None,
    description=None,
    default_value=0.0,
    min_value=None,
    max_value=None,
    since_version=None,
)
```

- Definition within a node/parameter group class:

```
learning_rate = knext.DoubleParameter("Learning rate", "The learning rate for Adam.", 0.003, min_value=0.)
```

- Usage within the `execute` method of the node class:

```
optimizer = torch.optim.Adam(lr=self.learning_rate)
```

- `knext.StringParameter` for string parameters and single-choice selections:
  - Signature:

```
knext.StringParameter(
    label=None,
    description=None,
    default_value="",
    enum: List[str] = None,
    since_version=None,
)
```

- Definition within a node/parameter group class:

```
# as a text input field
search_term = knext.StringParameter("Search term", "The string to search for in the text.", "")

# as a single-choice selection
selection_param = knext.StringParameter("Selection", "The options to choose from.", "A", enum=["A", "B", "C", "D"])
```

- Usage within the `execute` method of the node class:

```
table[table["str_column"].str.contains(self.search_term)]
```

- `knext.BoolParameter` for boolean parameters:
  - Signature:

```
knext.BoolParameter(
    label=None,
    description=None,
    default_value=False,
    since_version=None,
)
```

  - Definition within a node/parameter group class:

```
output_image = knext.BoolParameter("Enable image output", "Option to output
the node view as an image.", False)
```

  - Usage within the `execute` method of the node class:

```
if self.output_image is True:
    # generate an image of the plot
```

- `knext.ColumnParameter` for a single column selection:
  - Signature:

```
knext.ColumnParameter(
    label=None,
    description=None,
    port_index=0, # the port from which to source the input table
    column_filter: Callable[[knext.Column], bool] = None, # a (lambda)
function to filter columns
    include_row_key=False, # whether to include the table Row ID column in
the list of selectable columns
    include_none_column=False, # whether to enable None as a selectable
option, which returns "<none>"
    since_version=None,
)
```

  - Definition within a node/parameter group class:


```
selected_col = knext.ColumnParameter(
    "Target column",
    "Select the column containing country codes.",
    column_filter= lambda col: True if "country" in col.name else False,
    include_row_key=False,
    include_none_column=True,
)
```

- Usage within the `execute` method of the node class:

```
if self.selected_column != "<none>":
    column = input_table[self.selected_column]
    # do something with the column
```

- `knext.MultiColumnParameter` for a multiple column selection
  - Signature:

```
knext.MultiColumnParameter(
    label=None,
    description=None,
    port_index=0, # the port from which to source the input table
    column_filter: Callable[[knext.Column], bool] = None, # a (lambda)
function to filter columns
    since_version=None,
)
```

  - Definition within a node/parameter group class:

```
selected_columns = knext.MultiColumnParameter(
    "Filter columns",
    "Select the columns that should be filtered out."
)
```

  - Setup within the `configure` method of the node class:

```
# the multiple column selection parameter needs to be provided the list of
columns of an input table
self.selected_columns = input_schema_1.column_names
```

  - Usage within the `execute` method of the node class:

```
for col_name in self.selected_columns:
    # drop the column from the table
```

All of the above have arguments `label` and `description`, which are displayed in the node description in KNIME Analytics Platform, as well as in the configuration dialog itself. Additionally, all parameter classes have an optional argument `since_version`, which can be used to specify the version of the extension that the parameter was introduced in. Please refer to the Versioning your extension section below for a more detailed overview.

Parameters are defined in the form of class attributes inside the node class definition (similar to Python descriptors):

```
@knext.node(…)
…
class MyNode:
    num_repetitions = knext.IntParameter(
        label="Number of repetitions",
        description="How often to repeat an action",
        default_value=42
    )

    def configure(…):
        …

    def execute(…):
        …
```

While each parameter type listed above has default type validation, they also support custom validation via a property-like decorator notation. By wrapping a function that receives a tentative parameter value, and raises an exception should some condition be violated, with the `@some_param.validator` decorator, you are able to add an additional layer of validation to the parameter `some_param`. This should be done *below* the definition of the parameter for which you are adding a validator, and *above* the `configure` and `execute` methods:

```
@knext.node(…)
...
class MyNode:
    num_repetitions = knext.IntParameter(
        label="Number of repetitions",
        description="How often to repeat an action",
        default_value=42
    )

    @num_repetitions.validator
    def validate_reps(value):
        if value > 100:
            raise ValueError("Too many repetitions!")

    def configure(…):
        ...

    def execute(…):
        ...
```

It is also possible to define groups of parameters, which are displayed as separate sections in the configuration dialog UI. By using the `@knext.parameter_group` decorator with a dataclass-like class definition, you are able to encapsulate parameters and, optionally, their validators into a separate entity outside of the node class definition, keeping your code clean and maintainable. A parameter group is linked to a node just like an individual parameter would be:


```
@knext.parameter_group(label="My Settings")
class MySettings:
    name = knext.StringParameter("Name", "The name of the person", "Bario")

    num_repetitions = knext.IntParameter("NumReps", "How often do we repeat?", 1,
min_value=1)

    @num_repetitions.validator
    def reps_validator(value):
        if value == 2:
            raise ValueError("I don't like the number 2")


@knext.node(…)
…
class MyNodeWithSettings:
    settings = MySettings()

    def configure(…):
        …

    def execute(…):
        …
```

Another benefit of defining parameter groups is the ability to provide group validation. As opposed to only being able to validate a single value when attaching a validator to a parameter, group validators have access to the values of all parameters contained in the group, allowing for more complex validation routines.

We provide two ways of defining a group validator, with the `values` argument being a dictionary of `parameter_name` : `parameter_value` mappings:

1. by implementing a `validate(self, values)` method inside the parameter group class definition:

   ```
   @knext.parameter_group(label=''My Group'')
   class MyGroup:
       first_param = knext.IntParameter(''Simple Int'',''Testing a simple int
   param'', 42)
       second_param = knext.StringParameter("Simple String", "Testing a simple string
   param", "foo")

    def validate(self, values):
        if values["first_param"] < len(values["second_param"]):
            raise ValueError("Params are unbalanced!")
   ```


2. by using the familiar `@group_name.validator` decorator notation with a validator function inside the class definition of the ``parent" of the group (e.g. the node itself, or a different parameter group):

```
@knext.parameter_group(label=``My Group'')
class MyGroup:
    first_param = knext.IntParameter(``Simple Int'', ``Testing a simple int
param'', 42)
    second_param = knext.StringParameter("Simple String", "Testing a simple string
param", "foo")

@knext.node(…)
…
class MyNode:
    param_group = MyGroup()

    @param_group.validator
    def validate_param_group(values):
        if values["first_param"] < len(values["second_param"]):
            raise ValueError("Params are unbalanced!")
```

> If you define a validator using the first method, and then define another validator for the same group using the second method, the second validator will **override** the first validator. If you would like to keep **both** validators active, you can pass the optional `override=False` argument to the decorator: `@param_group.validator(override=False)`.

Intuitively, parameter groups can be nested inside other parameter groups, and their parameter values accessed during the parent group’s validation:

```
@knext.parameter_group(label="Inner Group")
class InnerGroup:
    inner_int = knext.IntParameter("Inner Int", "The inner int param", 1)

@knext.parameter_group(label="Outer Group")
class OuterGroup:
    outer_int = knext.IntParameter("Outer Int", "The outer int param", 2)
    inner_group = InnerGroup()

    def validate(self, values):
        if values["inner_group"]["inner_int"] > values["outer_int"]:
            raise ValueError("The inner int should not be larger than the outer!")
```


Node view declaration

You can use the `@knext.output_view(name="", description="")` decorator to specify that a node returns a view. In that case, the `execute` method should return a tuple of port outputs and the view (of type `knime.api.views.NodeView`).

```
from typing import List
import knime.extension as knext
import seaborn as sns


@knext.node(name="My Node", node_type=knext.NodeType.VISUALIZER, icon_path="icon.png",
category="/")
@knext.input_table(name="Input Data", description="We read data from here")
@knext.output_view(name="My pretty view", description="Showing a seaborn plot")
class MyViewNode:
    """
    A view node

    This node shows a plot.
    """

    def configure(self, config_context, input_table_schema)
        pass

    def execute(self, exec_context, table):
        df = table.to_pandas()
        sns.lineplot(x="x", y="y", data=df)
        return knext.view_seaborn()
```

Accessing flow variables

You can access the flow variables available to the node in both the `configure` and `execute` methods, via the `config_context.flow_variables` and `exec_context.flow_variables` attributes respectively. The flow variables are provided as a dictionary of `flow_variable_name` : `flow_variable_value` mappings, and support the following types:

- bool
- list(bool)
- float
- list(float)
- int

- list(int)
- str
- list(str)

By mutating the `flow_variables` dictionary, you can access, modify, and delete existing flow variables, as well as create new ones to be propagated to downstream nodes.

Versioning your extension

As you continue to develop your extension after the initial release, you might extend the functionality of your nodes by adding or removing certain parameters. With the versioning capabilities of Python-based node extensions for KNIME Analytics Platform, you can ensure backward compatibility for your users.

As seen in the Python Node Extension Setup section, the `knime.yml` configuration file contains a `version` field. This allows you to assign a version to each iteration of your extension. How closely you want to follow the semantic versioning scheme is completely up to you, but we do require adherence to the following formatting-related rule: versions must be composed of three non-negative numeric parts separated by dots (e.g. `1.0.0`, `0.2.1`, etc.).

> The version numbers are compared from left to right, i.e. `1.0.1` is newer than `1.0.0`, but older than `1.1.0`.

When adding a new parameter to a node, you should associate it with the corresponding version of your extension. This is done using the `since_version` argument that is now available for all parameter types via the appropriate constructor (e.g. `knext.IntParameter`), as well as parameter groups via the `@knext.parameter_group` decorator. If not specified, the `since_version` argument of a parameter or parameter group defaults to `0.0.0`, which indicates that the parameter was available from the first iteration of the extension.

A common use-case of extension versioning is to facilitate backward compatibility when opening workflows that were created/saved with an older version of the extension installed on the machine. What KNIME Analytics Platform will try to achieve by default in this case, is to combine the values of the previously configured node settings that are still available in the current version of the extension with the newly added node settings, if any. The latter are then automatically set to their default values, and the node remains configured.

> Sometimes the default value for a newly added node should be different than the default value for a node that is loaded as part of an old workflow (for an example see `double_param` below). In this scenario you can use a `DefaultValueProvider` instead of the default value. The `DefaultValueProvider` is a function that given a `Version` produces the default value of the parameter for that version of the extension. For old workflows it is called with the extension version the workflow was saved with. For new workflows it is called with the current version of the extension.

Here is a minimal functional example of a Python-based extension containing a single node with a single parameter. Since the parameter is available from the initial release of the extension, we can forgo setting the `since_version` argument:

```
"""
My Extension | Version: 0.1.0 | Author: Jane Doe
"""

import knime.extension as knext

@knext.node(
    "My Node",
    knext.NodeType.SOURCE,
    "../icons/icon.png",
    "/"
)
@knext.output_table("Output Data", "Data generated by this node.")
class MyNode:
    """Short node description.
    Long node description.
    """
    my_param = knext.IntParameter(
        "My Param",
        "My int parameter.",
        42,
    )

    def configure(self, config_context, input_table_schema):
        return input_table_schema

    def execute(self, exec_context, input_table):
        df = input_table.to_pandas()
        df['column1'] += self.my_param
        return knext.Table.from_pandas(df)
```

During the next few releases of the extension, `MyNode` is modified with an addition of several new parameters:


```
"""
My Extension | Version: 0.5.0 | Author: Jane Doe
"""

import knime.extension as knext

@knext.node(
    "My Node",
    knext.NodeType.SOURCE,
    "../icons/icon.png",
    "/"
)
@knext.output_table("Output Data", "Data generated by this node.")
class MyNode:
    """Short node description.
    Long node description.
    """
    my_param = knext.IntParameter(
        "My Param",
        "My int parameter.",
        42,
    )
    double_param = knext.DoubleParameter(
        "My Double",
        "Double parameter that strives to be Pi.",
        # For old workflows the value must be 1 to stay backwards compatible
        # but for new workflows we want the default to be 3.14
        lambda v: 1 if v < knext.Version(0, 3, 0) else 3.14,
        since_version="0.3.0",
    )
    string_param = knext.StringParameter(
        "My String",
        "An important string parameter to be turned into a flow variable.",
        "Foo",
        since_version="0.5.0",
    )

    def configure(self, config_context, input_table_schema):
        return input_table_schema

    def execute(self, exec_context, input_table):
        df = input_table.to_pandas()
        df['column1'] += self.my_param * self.double_param
        exec_context.flow_variables['important_string'] = self.string_param
        return knext.Table.from_pandas(df)
```

Now, if a user whose version of `My Extension` is `0.5.0` opens a workflow containing `MyNode` that was configured/saved on a machine where the version of `My Extension` was, for instance, `0.2.0`, the node settings will automatically be adapted to contain the previously configured value for `my_param`, and the default values for `double_param` and `string_param`. If the user were to execute the node without first reconfiguring it, the `execute` method would use those default values for the corresponding parameters.

Note how the default value of `double_param` depends on the version in order to ensure that the node's output does not change if the workflow is of an older version.

If the behaviour/functionality of the node has changed throughout the various releases of the extension, and you would like to require users to reconfigure the node if certain conditions are met, you can use the `config_context.set_warning()` or `exec_context.set_warning()` methods in the `configure` and `execute` methods of your node respectively to display a yellow "warning" sign in the node status. Additionally, you can raise an exception to further direct the user to reconfigure the node. For example:

```
import knime.extension as knext

@knext.node(
    "My Node",
    knext.NodeType.SOURCE,
    "../icons/icon.png",
    "/"
)
@knext.output_table("Output Data", "Data generated by this node.")
class MyNode:
    """Short node description.
    Long node description.
    """
    my_param = knext.IntParameter(
        "My Param",
        "My int parameter.",
        42,
    )
    double_param = knext.DoubleParameter(
        "My Double",
        "Double parameter that strives to be Pi.",
        lambda v: 1 if v < knext.Version(0, 3, 0) else 3.14,
        since_version="0.3.0",
    )

    def configure(self, config_context, input_table_schema):
        if self.my_param < 10:
            config_context.set_warning("Please reconfigure the node.")
            raise ValueError("My Param cannot be less than 10.")

        return input_table_schema

    def execute(self, exec_context, input_table):
        df = input_table.to_pandas()
        df['column1'] += self.my_param * self.double_param
        return knext.Table.from_pandas(df)
```

Deprecation of nodes

Sometimes it is not possible to change a node and stay backwards compatible e.g. if an input or output port is added. If you find yourself in this scenario do the following:

- Deprecate the old node by setting the `is_deprecated` argument to `true` in the `knime.extension.node` decorator. The node is then no longer listed in the node repository but it can still be loaded in existing KNIME workflows in which it then is also marked as deprecated.

- Implement a new version of the node that has the same `name` argument in the `knime.extension.node` decorator as the old node.

> Don’t change the name of the Python class that implements your old node because this name is used as ID by the Analytics Platform to find the node.

Improving the node description with Markdown

The description of your node, which is displayed in the *Description* area of KNIME Analytics Platform when a node is selected, is composed of multiple components. These components come from the descriptions you, as the developer, provide when defining the building blocks of the node, such as the input ports or the configuration parameters.

> Keep in mind that at the first line of the description docstring, next to the three double quotes, you can provide a short description, which will be shown in the overview when clicking on a category in the node repository of the KNIME Analytics Platform.

By including the `markdown` Python package in the `conda` environment associated with your node extension, you can make use of Markdown syntax when writing these descriptions to improve readability and the overall look of your nodes' documentation.

Below you can find a list of which Markdown syntax is supported for each node description element.

> As KNIME Analytics Platform transitions to the Modern UI, we will work on extending our support for additional Markdown syntax.

Table 1. The supported Markdown syntax for the available node description components

Element	Node description	Port description	Parameter description	Top-level parameter group description
Heading	✓	⌧	⌧	⌧
Bold	✓	✓	✓	✓
Italic	✓	✓	✓	✓

Element	Node description	Port description	Parameter description	Top-level parameter group description
Ordered List	✓	✓	✓	✓
Unordered List	✓	✓	✓	✓
Code	✓	✓	✓	✓
Fenced code blocks	✓	✓	✓	⊠
Horizontal Rule	✓	✓	⊠	⊠
Link	✓	✓	✓	✓
Table	✓	⊠	⊠	⊠

Here is a functional example of using Markdown when writing a Python node:

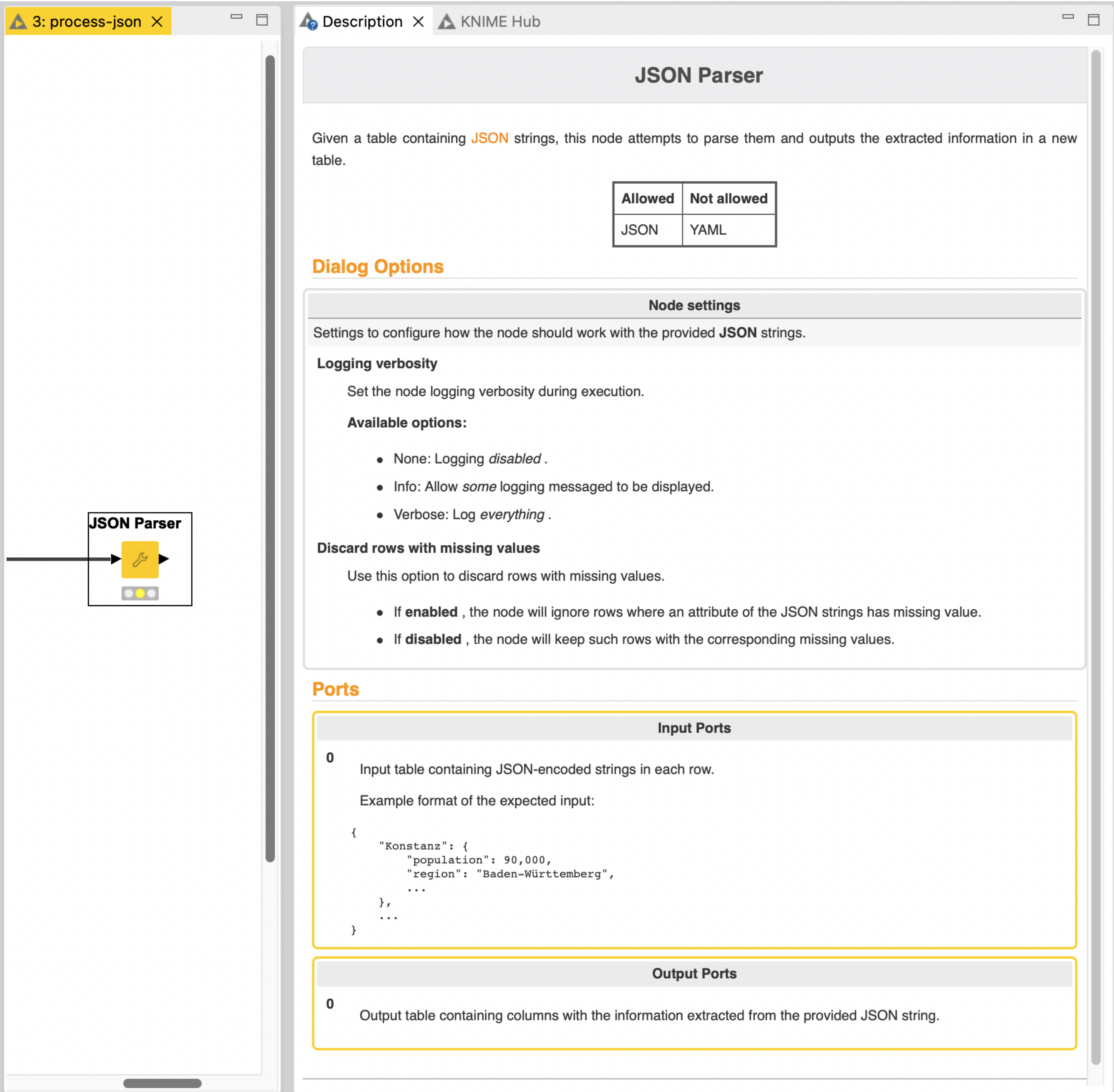
```
import knime.extension as knext

@knext.parameter_group("Node settings")
class Settings:
    """
    Settings to configure how the node should work with the provided **JSON** strings.
    """

    class LoggingOptions(knext.EnumParameterOptions):
        NONE = ("None", "Logging *disabled*.")
        INFO = ("Info", "Allow *some* logging messaged to be displayed.")
        VERBOSE = ("Verbose", "Log *everything*.")

    logging_verbosity = knext.EnumParameter(
        "Logging verbosity",
        "Set the node logging verbosity during execution.",
        LoggingOptions.INFO.name,
        LoggingOptions,
    )
```

```
    discard_missing = knext.BoolParameter(
        "Discard rows with missing values",
        """
        Use this option to discard rows with missing values.

        - If **enabled**, the node will ignore rows where an attribute of the JSON
strings has missing value.
        - If **disabled**, the node will keep such rows with the corresponding missing
values.
        """,
        True,
    )


@knext.node("JSON Parser", knext.NodeType.MANIPULATOR, "icon.png", main_category)
@knext.input_table(
    "Input table",
    """
    Input table containing JSON-encoded strings in each row.

    Example format of the expected input:
    ```
    {
        "Konstanz": {
            "population": 90,000,
            "region": "Baden-Württemberg",
            ...
        },
        ...
    }
    ```
    """,
)
@knext.output_table(
    "Parsed JSON",
    "Output table containing columns with the information extracted from the provided
JSON string.",
)
class JsonParser:
    """Node for parsing JSON strings.
    Given a table containing [JSON](https://developer.mozilla.org/en-
US/docs/Glossary/JSON) strings, this node attempts to parse them and
    outputs the extracted information in a new table.

    | Allowed | Not allowed |
    | ------- | ----------- |
    | JSON | YAML |
    """

    settings = Settings()

    def configure(self, config_context, input_table_schema):
```

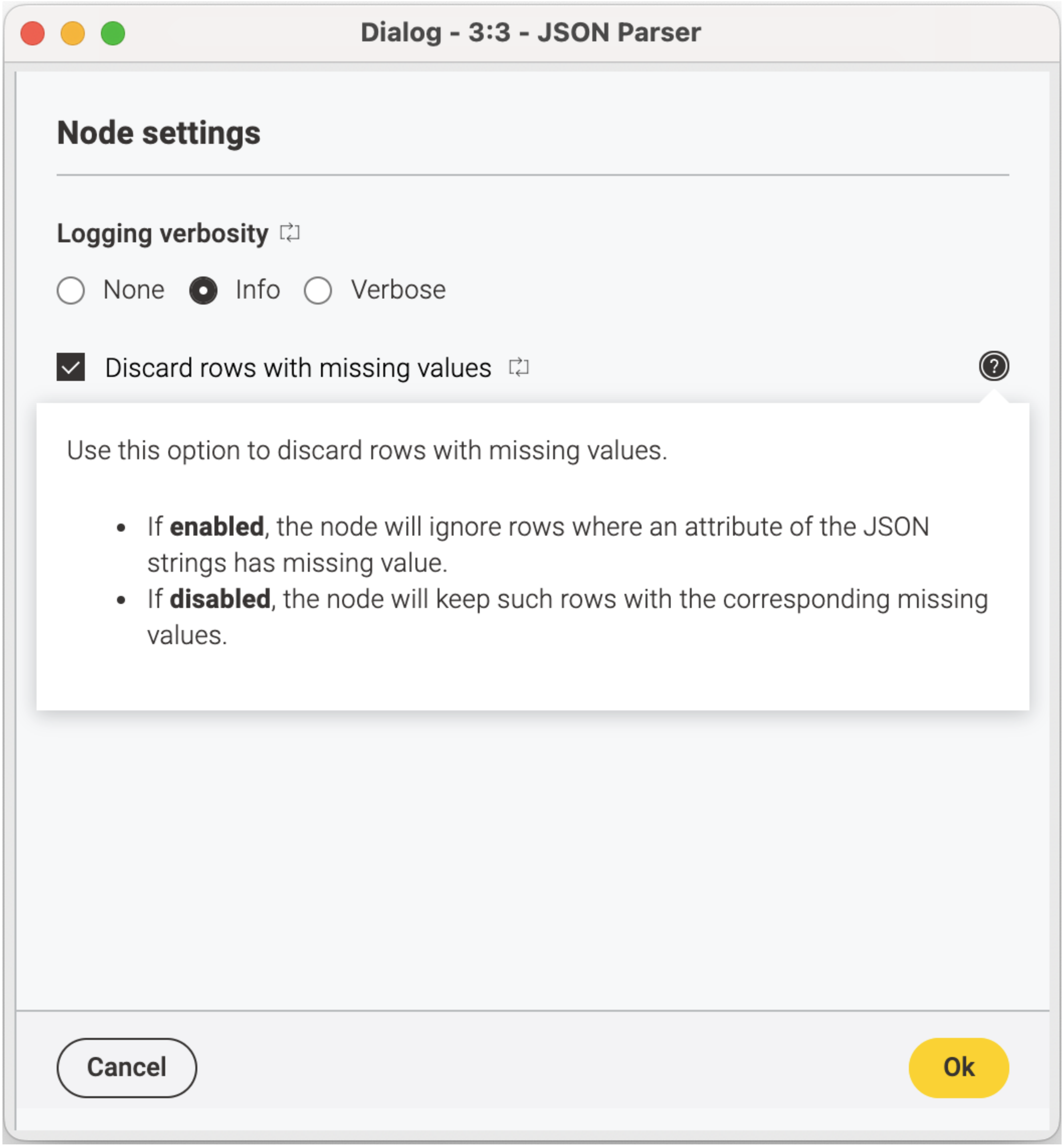


```
        # configuration routine
        # ...
        return input_table_schema

    def execute(self, exec_context, input_table):
        # execution routine
        # ...
        return input_table
```

Below is the resulting node description as seen in KNIME Analytics Platform:



The descriptions of individual node parameters can additionally be accessed from within the configuration dialog of the node:


Dialog - 3:3 - JSON Parser
Node settings
Logging verbosity
None
Info
Verbose
Discard rows with missing values
Use this option to discard rows with missing values.
If enabled, the node will ignore rows where an attribute of the JSON strings has missing value.
If disabled, the node will keep such rows with the corresponding missing values.
Cancel
Ok

Share your extension

You can share your extension in two ways. One is to bundle the extension to get a local update site which can be shared with your team or used for testing. The other is to publish it on KNIME Hub and make it available for the community. Either of the two options need some setup details. In this section, the setup and the two options will be explained.

Setup

To ensure that the users you have shared your extension with are able to utilise its functionality fully and error-free, we bundle the source files together with the required packages using `conda` as the bundling channel.

The `knime.yml` file (refer to the Python Node Extension Setup section for an example of this configuration file) contains the information required to bundle your extension, including:

- `extension_module`: the name of the `.py` file containing the node definitions of your extension.
- `env_yml_path`: the path to the `.yml` file containing the configuration of the `conda` environment that is used with your extension (see example below).

These YAML files can be automatically generated by activating the desired environment, and running one of the following commands, which will result in configuration files of various strictness:

- `conda env export > <env_yml_filename.yml>`, which will contain all the dependencies with their full version and build numbers. **Not recommended**
- `conda env export --from-history > <env_yml_filename.yml>`, which will reduce the list of dependencies down to the packages that you have manually installed in the environment. Note that this option does not preserve the list of manually specified channels when installing packages (e.g. *conda-forge*), so you might have to add them yourself.
- `conda env export | cut -f 1 -d "=" | grep -v "prefix" > <env_yml_filename.yml>`, which will preserve the list of custom channels used when installing packages, as well as the full list of dependencies, **without** strict versions specified.
- `conda env export --no-builds | grep -v "prefix" > <env_yml_filename.yml>`, same as the above command, but with package versions specified (excluding build numbers).

Note that, in addition to packages installed with `conda`, you are also able to install packages from *PyPI* via `pip`. When done from within your activated `conda` environment, such packages are also automatically included in the YAML configuration file generated with the above commands.

`environment.yml`:

```
name: knime-python-scripting
channels:
- conda-forge
- knime
dependencies:
- python=3.9              # base dependency
- knime-python-base>=4.7  # base dependency
- knime-extension>=4.7    # base dependency
- <another-package>=1.0.1
- <yet-another-package>
- pip:
    - img2text
    - pillow
```

OS-specific environments

Since KNIME Analytics Platform is available on Windows, Linux, and macOS, you should try your best to ensure that your Python extension performs as expected on all platforms. To achieve this, you can generate OS-specific YAML files that include versions/replacements of packages that are guaranteed to be available on this particular OS by, for instance, searching the Anaconda package repository with the *Platform* filter set to the desired OS (e.g. `osx-64` for Intel-based Mac machines), and correspondingly building the environment YAML file.

When specifying the environment YAMLs in the `knime.yml` file of your Python extension, you can use the following format to include different environment configuration files for different operating systems, which the KNIME Analytics Platform will then appropriately use together with your extension:

> Support for Apple Silicon-specific environments is available starting from the 4.7 release of KNIME Analytics Platform.

`knime.yml`:


```
...
env_yml_path:
  osx-64: <env_for_intel_mac>
  osx-arm64: <env_for_arm_mac> # available starting from KNIME Analytics Platform 4.7
  linux-64: <env_for_linux>
  win-64: <env_for_win>
...
```

Lastly, a new extension needs a `LICENSE.TXT` that will be displayed during the installation process.

Option 1: Bundling a Python extension to share a zipped update site

Once you have finished implementing your Python extension, you can bundle it, together with the appropriate `conda` environment, into a local update site. This allows other users to install your extension in the KNIME Analytics Platform.

Follow the steps of `extension setup`. Once you have prepared the YAML configuration file for the environment used by your extension, and have set up the `knime.yml` file, you can proceed to generating the local update site.

We provide a special `conda` package, `knime-extension-bundling`, which contains the necessary tools to automatically build your extension. Run the following commands in your terminal (Linux/macOS) or Anaconda Prompt (Windows). They will setup a `conda` environment, which gives the tools to bundle extensions. Then the extension will be bundled.

1. Create a fresh environment prepopulated with the `knime-extension-bundling` package:

   ```
   conda create -n knime-ext-bundling -c knime -c conda-forge knime-extension-
   bundling
   ```

2. Activate the environment:

   ```
   conda activate knime-ext-bundling
   ```

3. With the environment activated, run the following command to bundle your Python extension:

   - macOS/Linux:

     ```
     build_python_extension.py <path/to/directoryof/myextension/>
     <path/to/directoryof/output>
     ```


- Windows:

```
build_python_extension.bat <path/to/directoryof/myextension/>
<path/to/directoryof/output>
```

where `<path/to/directoryof/myextension/>` is the path to the directory containing your `.py` extension module and the `knime.yml` file, and `<path/to/directoryof/output>` is the path to the directory where the bundled extension **repository** will be stored.

> By default, the `build_python_extension` command will bundle the extension for KNIME Analytics Platform 4.7 (this will include the Apple Silicon-specific environment, if provided). If you need to bundle the extension for KNIME Analytics Platform 4.6, you can do so by adding `--knime-version 4.6` to the command. The environment YAML files **must** contain the corresponding versions of the `knime-python-base` and `knime-extension` packages, e.g. `- knime-python-base=4.6` when bundling for version 4.6.

> The bundling process can take several minutes to complete.

4. Add the generated **repository** folder to KNIME AP as a Software Site in *File* → *Preferences* → *Install/Update* → *Available Software Sites*
5. Install it via *File* → *Install KNIME Extensions*

The generated repository can now be shared with and installed by other users.

Option 2: Publish your extension on KNIME Hub

Once you have finished implementing your Python extension, you can share it, together with the appropriate `conda` environment, to KNIME Hub.

Provide the extension

Follow the steps of `extension setup` to prepare the `environment.yml` or some other `yml` defining your Python environment and the `knime.yml`.

Upload your extension into a Git repository, where the `knime.yml` is found top-level. A `config.yml` is not needed.

Some recommended project structure:

```
https://github.com/user/my_knime_extension
├──── icons
│     └──── my_node_icon.png
├──── knime.yml
├──── LICENSE.txt
├──── environment.yml
└──── my_extension.py
```

Write a test workflow

1. Install the `KNIME Testing Framework` to your KNIME Analytics Platform (KAP)
2. Create a test workflow (see https://www.knime.com/automated-workflow-testing-and-validation for details)
3. Test your extension against the test workflow: does it check your functionality and behaves as expected?

Contribute

Send the link to your repository to community-contributions@knime.org. Additionally, request `community contributor status` for your forum account, which will allow you to upload test workflows to the *KNIME Community Server*.

Lean back, clean up

1. Wait for us to come back to you
2. If it is available on the nightly experimental community extension Hub, please test it again (with your test workflow) by using the nightly experimental update site: https://update.knime.com/community-contributions/trunk (for now, every Python extension will stay on that site)
3. Upload the test workflow onto the Community Workflow Server. You can access the server via the KNIME Explorer view. If you don't have a mount point entry for the community server yet, click on the button at the top-right of the view and then on *Configure Explorer settings* in the appearing dialog. Now create a new mount point with a custom ID and *KNIME Community Server* as mount point type. You can log into the server using your forum credentials, if you got your requested `community contributor status`. Create a new workflow group inside *Testflows/trunk*, give it a meaningful name, and finally upload your workflow(s) into this group. Please make sure that the

permissions on the group and the workflow(s) allow read access for everyone.

Customizing the Python executable

Some extensions might have additional requirements that are not part of the bundled environment e.g. in case of third party models. For these extensions, it is possible to overwrite the Python executable used for execution. This can be done via the system property `knime.python.extension.config` that has to point to a special YAML file on disc. Add it to your knime.ini with the following line:

```
-Dknime.python.extension.config=path/to/your/config.yml
```

> The forward slash `/` has to be used on all OS, also on Windows.

The format of the YAML is:

```
id.of.first.extension:
  conda_env_path: path/to/conda/env
id.of.second.extension:
  python_executable: path/to/python/executable
```

You have two options to specify a custom Python exectuable:

- Via the `conda_env_path` property (recommended) that points to a `conda` environment on your machine.
- Via the `python_executable` property that points to an executable script that starts Python (see Manually configured Python environments section in KNIME Python Integration Guide for more details).

If you specify both, then `conda_env_path` will take precedence. It is your responsibility to ensure that the Python you specified in this file has the necessary dependencies to run the extension. As illustrated above, you can overwrite the Python executable of multiple extensions.

Registering Python extensions during development

In order to register a Python extension you are developing, you can add it to the `knime.python.extension.config` YAML explained above by adding a src property:

```
id.of.your.dev.extension:
  src: path/to/your/extension
  conda_env_path: path/to/conda/env
  debug_mode: true
```

Note that you have to specify either `conda_env_path` or `python_executable` because the Analytics Platform doesn't have a bundled environment for your extension installed. For debugging it is also advisable to enable the debug mode by setting `debug_mode: true`. The debug mode disables caching of Python processes which allows some of your code changes to be immediately shown in the Analytics Platform. Those changes include:

- Changes to the execute and configure runtime logic.
- Changes to existing parameters e.g. changing the `label` argument.
- Other changes, such as adding a node or changing a node description, require a restart of the Analytics Platform to take effect.
- Last but not least, fully enabling and disabling the debug mode also requires a restart.

Other Topics

Logging

You can use the logging Python module to send warnings and errors to the KNIME Analytics Platform console. By going to *File* → *Preferences* → *KNIME* → *KNIME GUI*, you can choose the Console View Log Level. Each consecutive level includes the previous levels (i.e. `DEBUG` will also allow message from `INFO`, `WARN`, and `ERROR` to come through in the console, whereas `WARN` will only allow `WARN` and `ERROR` levels of messages).

In your Python script, you can initiate the logger, and use it to send out messages to the KNIME Analytics Platform console as follows:

```
# other various imports including knime.extension
import logging

LOGGER = logging.getLogger(__name__)

# your node definition via the knext decorators
class MyNode:
    # your configuration dialog parameter definitions

    def configure(…):
        ...
        LOGGER.debug("This message will be displayed in the KNIME Analytics Platform
console at the DEBUG level")
        LOGGER.info("This one will be displayed at the INFO level.")
        LOGGER.warning("This one at the WARN level.")
        LOGGER.error("And this will be displayed as an ERROR message.")
        ...

    def execute(…):
        ...
        LOGGER.info("Logger messages can be inserted anywhere in your code.")
        ...
```

Gateway caching

In order to allow for a smooth user experience, the Analytics Platform caches the gateways used for non-execution tasks (such as the spec propagation or settings validation) of the last used Python extensions. This cache can be configured via two system properties:

- `knime.python.extension.gateway.cache.size`: controls for how many extensions the gateway is cached. If the cache is full and a gateway for a new extension is requested, then the gateway of the least recently used extension is evicted from the cache. The default value is 3.

- `knime.python.extension.gateway.cache.expiration`: controls the time period in seconds after which an unused gateway is removed from the cache. The default is 300 seconds.

The `debug_mode: true` propertly of `config.yml` discussed before effectively disables caching for individual extensions. By default, all extensions use caching.

Open for Innovation
KNIME

KNIME AG
Talacker 50
8001 Zurich, Switzerland
www.knime.com
info@knime.com