KNIME Python Integration Guide

KNIME AG, Zurich, Switzerland

Version 4.5 (last updated on 2022-05-02)

Table of Contents

Introduction

This guide describes how to install and configure the KNIME Python Integration to be used with KNIME Analytics Platform.

In this guide, we refer to the KNIME Python Integration available since the v3.4 release of KNIME Analytics Platform, which supports Python 2 and 3. With the v4.5 release of KNIME Analytics Platform, we are making available the new Python Script (Labs) node, which provides a significantly more performant way of working with Python in KNIME Analytics Platform, and supports Python versions 3.6 - 3.9. This node is part of the KNIME Python Integration (Labs) extension.

The KNIME Python Integration relies on an existing installation of Python, and requires it to have certain packages. While there are many ways of installing and configuring Python environments, our recommended way is to use the Conda package manager. In this guide, we will describe how to install Python and the necessary packages using Conda, how to configure the KNIME Python Integration, as well as go through the available nodes and examine their functionality.

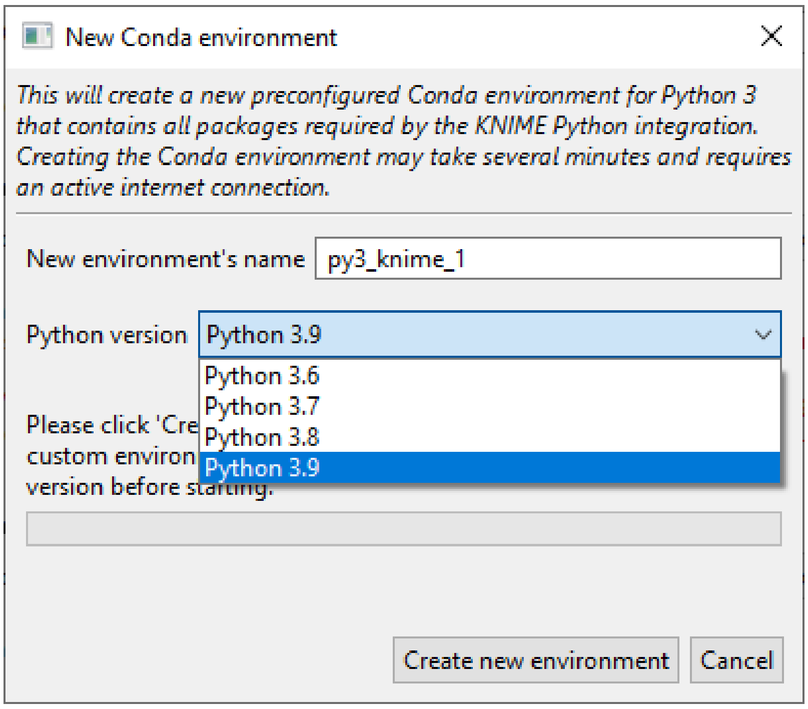
Quickstart

This quickstart guide goes through the basic steps required to install the KNIME Python Integration and its prerequisites. If you'd like a more thorough explanation, please refer to the sections that follow after this quickstart.

1. First, install the KNIME Python Integration extension. In KNIME Analytics Platform, go to *File* → *Install KNIME Extensions*. The KNIME Python Integration can be found under *KNIME & Extensions* or by entering *Python Integration* into the search box. Optionally, install the KNIME Python Integration (Labs) extension that contains the new Python Script (Labs) node as well.
2. Next, install a distribution of the Conda package manager, for example Miniconda. It comes with Python included, and is used to manage Python packages and environments.
3. With Conda and Python installed, go to the Python Preference page located at *File* → *Preferences*. Select *KNIME* → *Python* from the list on the left. In the page that opens, select **Conda** under *Python environment configuration*. Next, provide the path to your Conda installation folder (for Miniconda, the default installation path for Windows is `C:\Users\<your-username>\miniconda3\`, for Mac: `/Users/<your-`

username>/miniconda3`, and Linux: `/home/<your-username>/miniconda3`). Once a valid path has been entered, the Conda version number will be shown.

4. Below the Conda version number you can choose which Conda environment is to be used for Python 3 and Python 2 by selecting it from a combo box. In case you have already set up an environment containing all the necessary dependencies for the KNIME Python Integration, just select it from the list and you are ready to go. If you do not have a suitable environment available, click the **New environment…** button, which will open the following dialog:



This will create a new preconfigured Conda environment for Python 3 that contains all packages required by the KNIME Python integration. Creating the Conda environment may take several minutes and requires an active internet connection.

New environment's name: py3_knime_1

Python version: Python 3.9

Python 3.6
Python 3.7
Python 3.8
Python 3.9

Please click 'Cre
custom environ
version before starting.

Provide a name for the new environment, choose the Python version you want to use, and click the **Create new environment** button. This creates a new Conda environment containing all the required dependencies for the KNIME Python Integration.

Depending on your internet connection, the environment creation may take a while as all packages need to be downloaded and extracted.

Once the environment is successfully created, the dialog closes and the new environment is selected automatically.

Installing Python with Conda

This section describes how to install and configure Python to be used with KNIME Python Integration. We recommend using Conda, which is a package and environment manager that simplifies the process of working with multiple versions of Python and different sets of packages by encapsulating them in so-called Conda environments. A Conda environment is essentially a folder that contains a specific Python version and the installed packages. This means you can have several different Python versions installed on your system at the same time in a clean and easy-to-maintain manner. When used with KNIME Analytics Platform, this is especially useful, as it allows you to use Python 3 and Python 2 at the same time without running into version issues. Furthermore, Conda is able to create predefined environments with a single command and makes it easy to add Python packages to existing ones.

There are different flavours of Conda available. Miniconda, for instance, is a minimal installation of the package and environment manager, together with your chosen version of Python. Note that after installation of Miniconda, only the `base` environment will contain that version of Python, and you will be able to create Conda environments configured with any version of Python that you would like to specify.

Indeed, we discuss the various ways of setting up Conda environments to include the dependencies needed for KNIME Python Integration in the Configure and manage Python environments section below.

With Python installed, we can now proceed to Setting up the KNIME Python Integration.

Setting up the KNIME Python Integration

This section describes how to install and configure the KNIME Python Integration using an existing installation of Python. We recommend using the Conda package and environment manager, which includes Python, and makes the set up process straightforward. If you haven't yet installed Python with Conda, please refer to the Installing Python with Conda section.

Note that you can also bypass using Conda altogether and configure the KNIME Python Integration with corresponding Python environments manually, which we will also cover below.

Installing the extension

From KNIME Analytics Platform, go to *File* → *Install KNIME Extensions* and search for *Python Integration*. The KNIME Python Integration extension should appear in the list. You can then select the extension and proceed through the installation wizard.

Configuring the KNIME Python Integration

Configure and manage Python environments

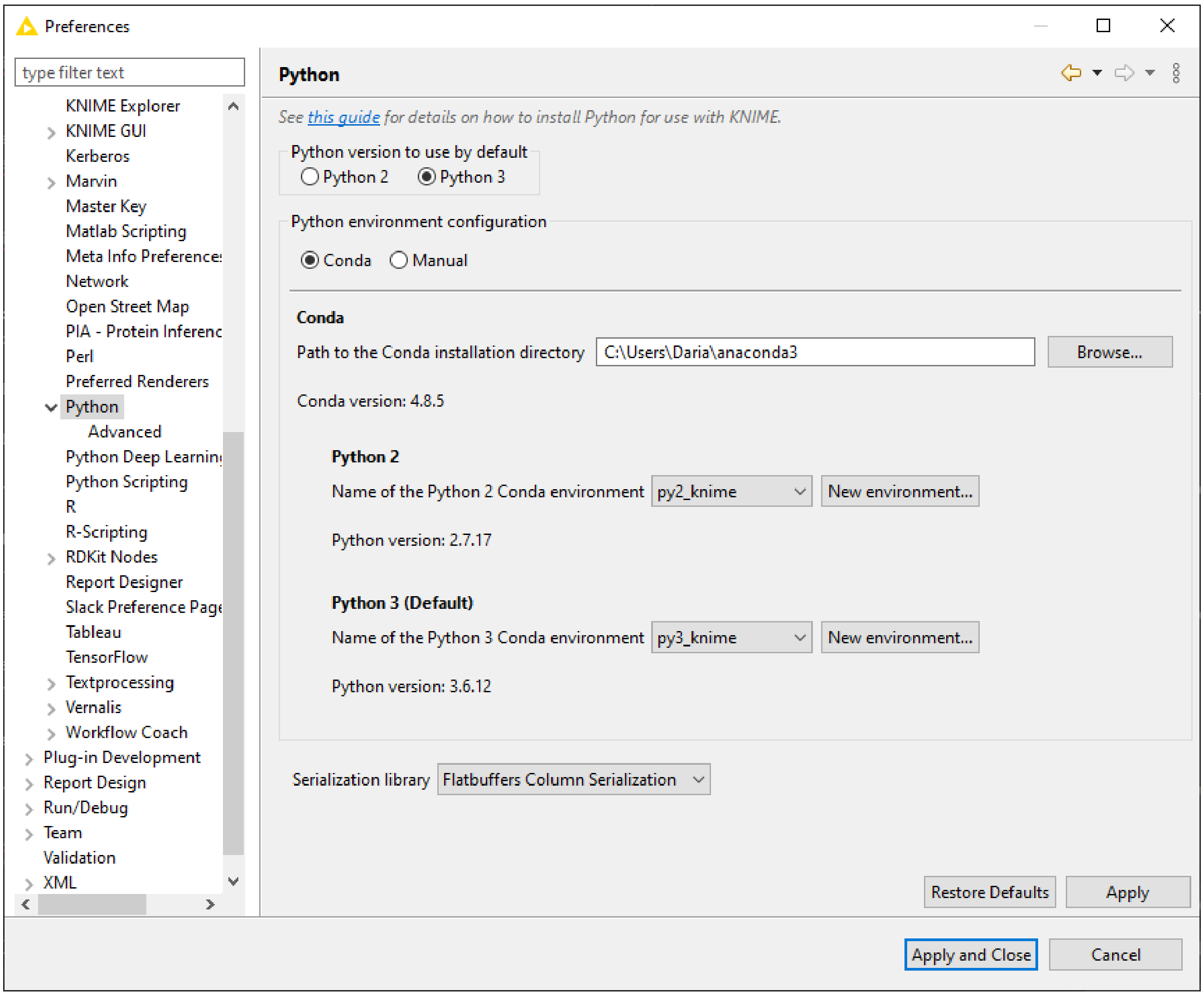
With the extension installed, we now need to set up the appropriate Python environments and configure KNIME Analytics Platform to use them. Navigate to the Preferences page for the KNIME Python Integration by going to *File* → *Preferences*, and then selecting *KNIME* → *Python* from the list on the left. The page will present you with different options for configuring the Python environment, namely:

- Conda environments:
  - Automatic via the Preference dialog (recommended)
  - Manual via YAML files
- Manually configured Python environments

Conda environments

Automatic (recommended)

Select **Conda** under *Python environment configuration*. The current page should look like the screenshot shown below.


Preferences

Python

See this guide for details on how to install Python for use with KNIME.

Python version to use by default: ○ Python 2 [✓ Python 3]

Python environment configuration: [✓ Conda] ○ Manual

Conda

Path to the Conda installation directory: C:\Users\Daria\anaconda3

Conda version: 4.8.5

Python 2

Name of the Python 2 Conda environment: py2_knime

Python version: 2.7.17

Python 3 (Default)

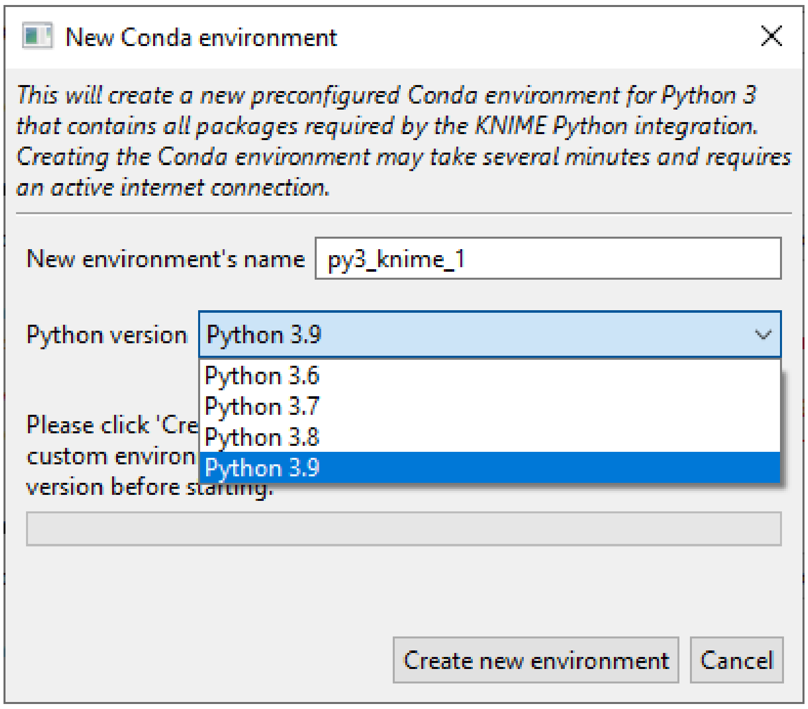
Name of the Python 3 Conda environment: py3_knime

Python version: 3.6.12

Serialization library: Flatbuffers Column Serialization

In the *Path to the Conda installation directory* field, provide the path to the folder containing your installation of Conda (for Miniconda, the default installation path is `C:\Users\<your-username>\miniconda3\` for Windows, `/Users/<your-username>/miniconda3` for Mac, and `/home/<your-username>/miniconda3` for Linux). Once you have entered a valid path, the installed Conda version will be displayed and KNIME Analytics Platform will automatically check for all available Conda environments.

Underneath the Conda version number, you can choose which environment should be used for Python 3 and Python 2 by selecting it from the corresponding combo box. If you have already set up a Python environment containing all the necessary dependencies for the KNIME Python Integration, just select it from the list and you are ready to go. Otherwise, click the **New environment…** button, which will open the following dialog:


New Conda environment
This will create a new preconfigured Conda environment for Python 3 that contains all packages required by the KNIME Python integration. Creating the Conda environment may take several minutes and requires an active internet connection.
New environment's name
py3_knime_1
Python version
Python 3.9
Python 3.6
Python 3.7
Python 3.8
Python 3.9
Please click 'Cre
custom environ
version before starting.
Create new environment
Cancel

Provide a name for the new environment, choose the Python version you want to use, and click the **Create new environment** button. This creates a new Conda environment containing all the required dependencies for the KNIME Python Integration. Refer to the Python version support section for details on which versions of Python are compatible with the KNIME Python Integration.

Depending on your internet connection, the environment creation may take a while as all packages need to be downloaded and extracted.

Once the environment is successfully created, the dialog will close and the new environment will be selected automatically. If everything went well, the Python version will be shown below the environment selection, and you are ready to go.

Manually create a Conda environment

If you do not want to create a Conda environment automatically from the Preferences page, you can create one manually using a YAML configuration file. Such files list all the important information about the Conda environment that will be created, such as the environment name, the packages to be installed, and the Conda channels where those packages are hosted. We have provided two such configuration files below (one configuration file to create a new Python 3 environment and one for Python 2). They list all of the dependencies needed for the KNIME Python Integration:

py3_knime.yml

```
name: py38_knime         # Name of the created environment
channels:                # Repositories to search for packages
- defaults
- anaconda
- conda-forge
dependencies:            # List of packages that should be installed
- python=3.8             # Python
- py4j                   # used for KNIME <-> Python communication
- nomkl                  # Prevents the use of Intel's MKL
- pandas                 # Table data structures
- jedi<=0.17.2           # Python script autocompletion
- python-dateutil        # Date and Time utilities
- numpy                  # N-dimensional arrays
- cairo                  # SVG support
- pillow                 # Image inputs/outputs
- matplotlib             # Plotting
- pyarrow=6.0            # Arrow serialization
- IPython                # Notebook support
- nbformat               # Notebook support
- scipy                  # Notebook support
- jpype1                 # Databases
- python-flatbuffers<2.0 # because tensorflow expects a version before 2
- h5py<3.0 # must be < 3.0 because they changed whether str or byte is returned
- protobuf>3.12
- libiconv               # MDF Reader node
- asammdf=5.19.14        # MDF Reader node
```

py2_knime.yml

```
name: py2_knime          # Name of the created environment
channels:                # Repositories to search for packages
- defaults
- anaconda
- conda-forge
dependencies:            # List of packages that should be installed
- python=2.7             # Python
- pandas=0.23            # Table data structures
- jedi=0.13              # Python script autocompletion
- parso=0.7.1            # Jedi dependency this is the last version compatible with 2.7
- python-dateutil=2.7 # Date and Time utilities
- numpy=1.15             # N-dimensional arrays
- cairo=1.14             # SVG support
- pillow=5.3             # Image inputs/outputs
- matplotlib=2.2         # Plotting
- pyarrow=0.11           # Arrow serialization
- IPython=5.8            # Notebook support
- nbformat=4.4           # Notebook support
- scipy=1.1              # Notebook support
- jpype1=0.6.3           # Databases
- protobuf=3.5           # Serialization for deprecated Python nodes
```

The above configuration files only contain the Python packages that the KNIME Python Integration depends on. If you want to use additional Python packages, you can either add the name of the package at the end of the configuration file or add them after the environment has been created.

For example, for Python 3 you can use the `py3_knime.yml` and download it to any folder on your system (e.g. your home folder). In order to create an environment from this file, open a *shell* (Linux), *terminal* (Mac), or *Anaconda prompt* (Windows, comes with Conda and can be found by entering `anaconda` in Windows Search), change the directory to the folder that contains the configuration file and execute the following command:

```
conda env create -f py3_knime.yml
```

This command creates a new environment with the name provided at the top of the configuration file (which you are welcome to change, of course). It also downloads and installs all of the listed packages (depending on your internet speed, this may take a while).

If you want to use both Python 3 and Python 2 at the same time, just repeat the above steps using the respective configuration file.

The list of dependencies for Python 3 and Python 2 is almost the same, however the version numbers change.

After Conda has successfully created the environment, Python is all set up and ready to go.

Further information on how to manage Conda environments can be found here.

Manually installing additional Python packages

The YAML configuration files listed above only contain the packages to be installed so that the KNIME Python Integration works properly. Hence, if you want to use Python packages other than the ones listed in the configuration files, these can be easily added manually after the environment has been created. E.g. if you want to use functionality from `scikit-learn` in KNIME Python nodes, you can use the following command in the command-line interpreter of your operating system:

```
conda install --name <ENV_NAME> scikit-learn
```

Just replace `<ENV_NAME>` with the name of the environment where you would like to install the package.

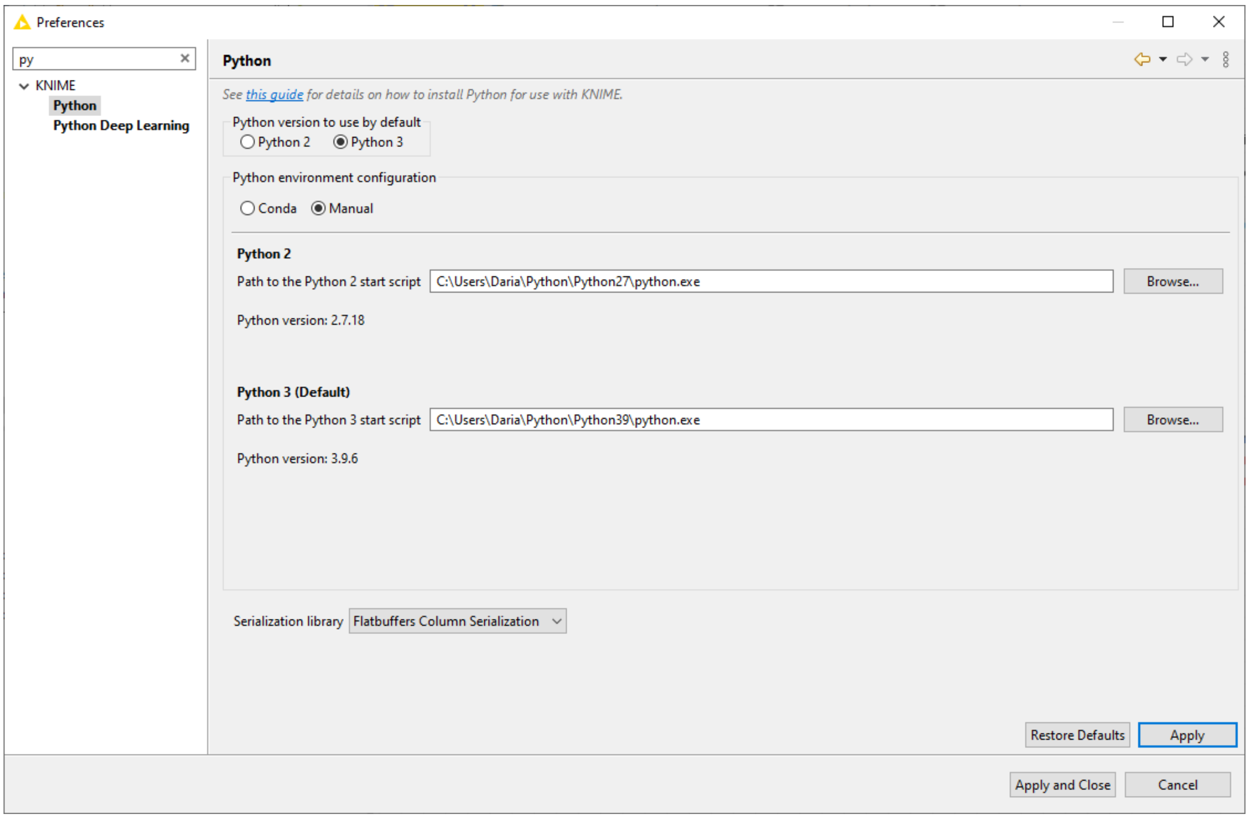
You can easily specify the version of the package with e.g. `scikit-learn==0.20.2`

Further information on how to manage Conda packages can be found here.

Manually configured Python environments

The alternative to using the Conda package and environment manager is to manually set up the Python installation. If you choose **Manual** under *Python environment configuration*, you will have the following options:

1. Point KNIME Analytics Platform to a Python executable of your choice



Preferences

py

KNIME
Python
Python Deep Learning

Python

See this guide for details on how to install Python for use with KNIME.

Python version to use by default: ○ Python 2 [✓ Python 3]

Python environment configuration: ○ Conda [✓ Manual]

Python 2

Path to the Python 2 start script: C:\Users\Daria\Python\Python27\python.exe

Python version: 2.7.18

Python 3 (Default)

Path to the Python 3 start script: C:\Users\Daria\Python\Python39\python.exe

Python version: 3.9.6

Serialization library: Flatbuffers Column Serialization

2. Point KNIME Analytics Platform to a start script which activates the environment you want to use for Python 2 and Python 3 respectively. This option assumes that you have created a suitable Python environment earlier with a Python virtual environment manager of your choice. In order to use the created environment for the KNIME Python Integration, you need to create a start script (shell script on Linux and Mac, batch file on Windows). The script has to meet the following requirements:

   - It has to start Python with the arguments given to the script (please make sure that spaces are properly escaped)
   - It has to output standard and error out of the started Python instance
   - It must not output anything else.

   Here we provide an example shell script for the Python environment on Linux and Mac. Please note that on Linux and Mac you additionally need to make the file executable (i.e. `chmod gou+x py3.sh`).

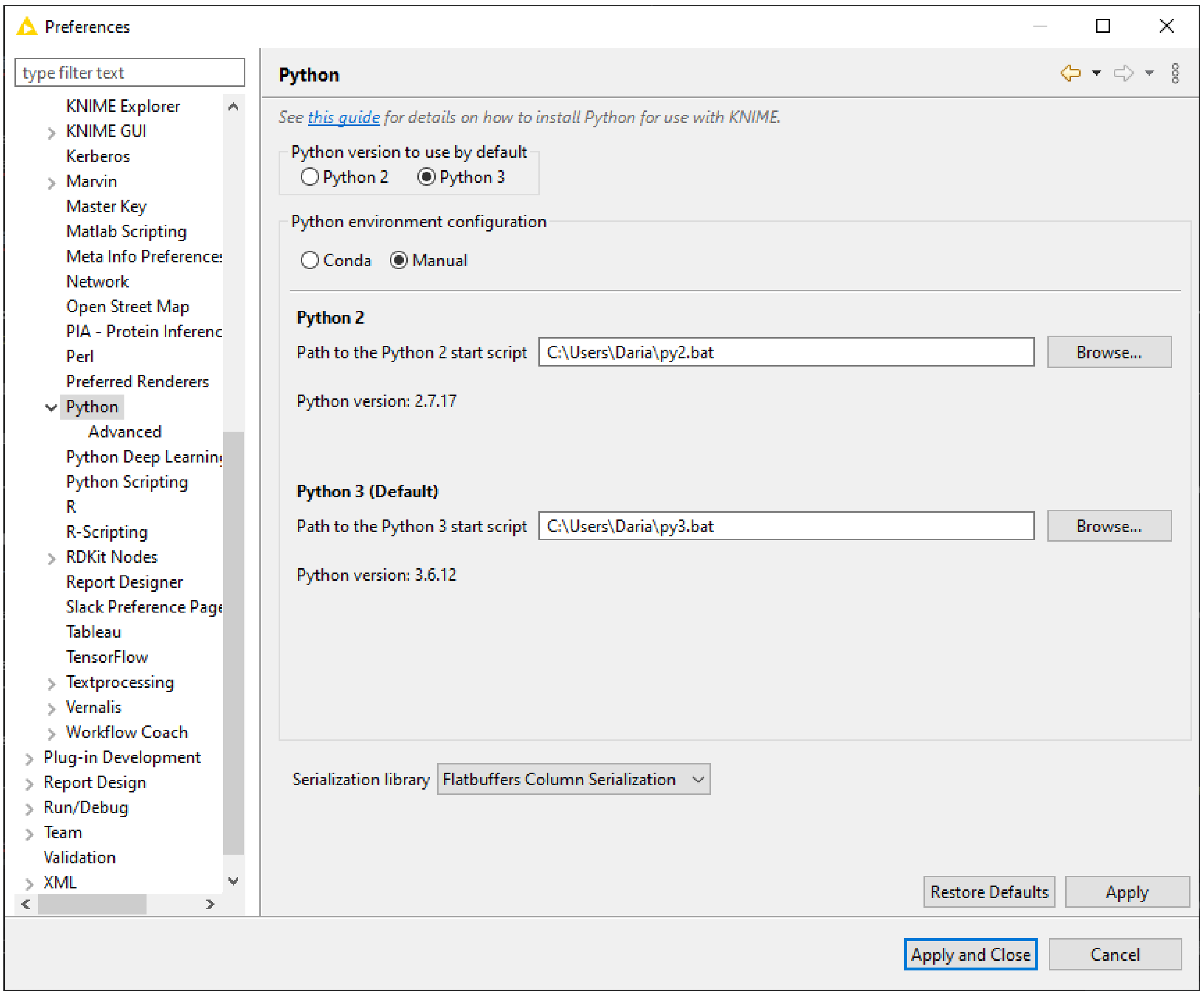
```
#! /bin/bash
# Start by making sure that the anaconda folder is on the PATH
# so that the source activate command works.
# This isn't necessary if you already know that
# the anaconda bin dir is on the PATH
export PATH="<PATH_WHERE_YOU_INSTALLED_ANACONDA>/bin:$PATH"

conda activate <ENVIRONMENT_NAME>
python "$@" 1>&1 2>&2
```

On Windows, the script looks like this:

```
@REM Adapt the folder in the PATH to your system
@SET PATH=<PATH_WHERE_YOU_INSTALLED_ANACONDA>\Scripts;%PATH%
@CALL activate <ENVIRONMENT_NAME> || ECHO Activating python environment failed
@python %*
```

These are example scripts for Conda. You may need to adapt them for other tools by replacing the Conda-specific parts. For instance, you will need to edit them in order to point to the location of your environment manager installation and to `activate` the correct environment.

After creating the start script, you will need to point KNIME Analytics Platform to it by specifying the path to the script on the Python Preferences page.



Python

See this guide for details on how to install Python for use with KNIME.

Python version to use by default: Python 2 [✓ Python 3]

Python environment configuration: Conda [✓ Manual]

Python 2
Path to the Python 2 start script: C:\Users\Daria\py2.bat
Python version: 2.7.17

Python 3 (Default)
Path to the Python 3 start script: C:\Users\Daria\py3.bat
Python version: 3.6.12

Serialization library: Flatbuffers Column Serialization

Figure 1. KNIME Python Preferences page. Here you can set the path to the executable script that launches your Python environment.

If you like, you can have configurations for both Python 2 and Python 3 (as is shown above). Just select the one that you would like to have as the default. If everything is set correctly, the Python version is now shown in the dialog window and you are ready to go.

Serialization library

You can choose which serialization library should be used by the KNIME Python Integration to transfer data from KNIME Analytics Platform to Python.

This option does not usually need to be changed and can be left as the default.

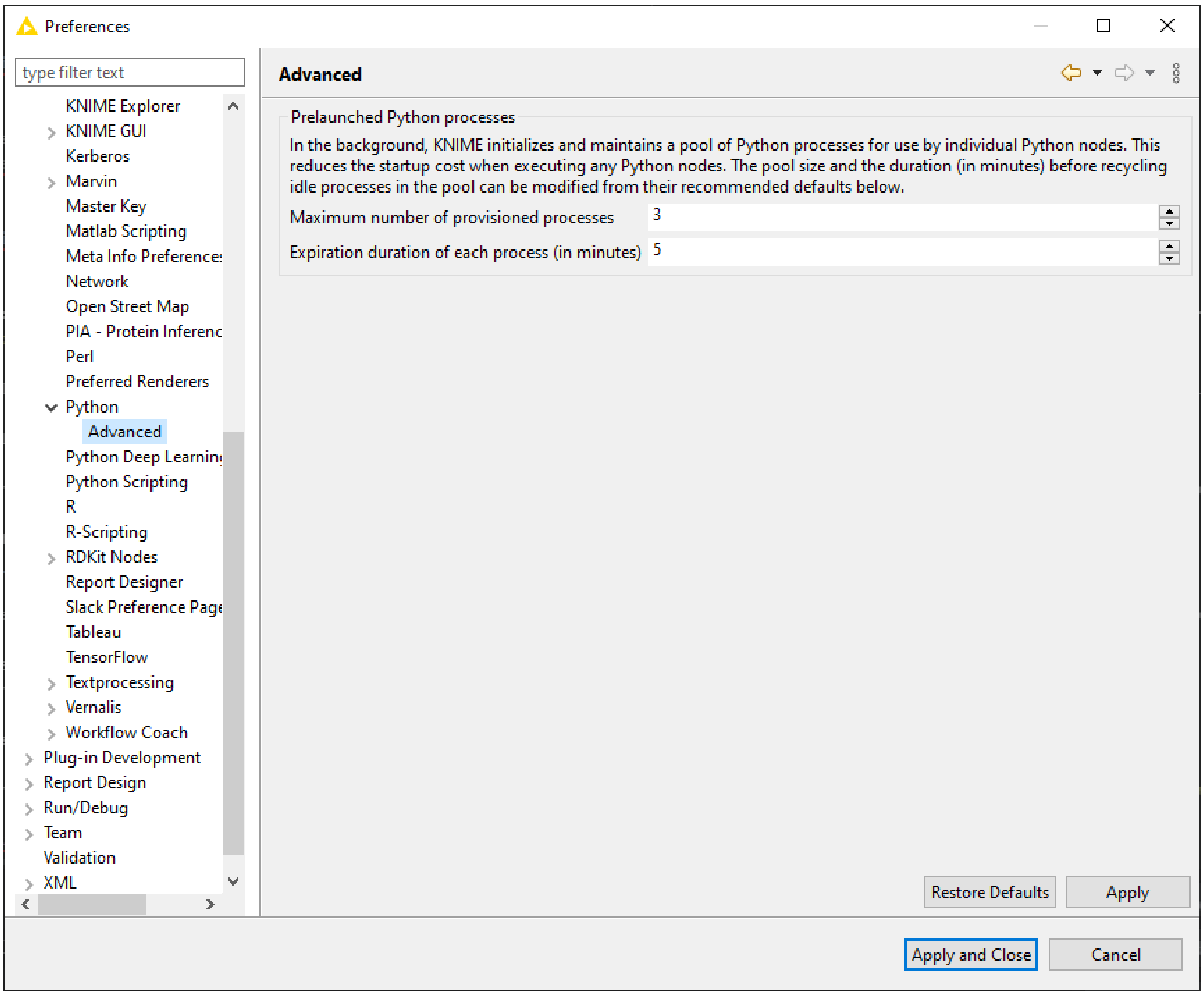
Some of these serialization libraries have additional dependencies stated below, however if you followed the automatic Conda environment set up, all required dependencies are already included (see the YAML configuration files for the required packages). Currently, there are three options:

- *Flatbuffers Column Serialization* (**default** & **recommended**): no additional dependencies
- *Apache Arrow*: provides a significant performance boost, depends on `pyarrow` version `4.0.1`
- *CSV (Experimental)*: depends on `pandas` version `0.23`

> Note that the serialization options do not apply to the KNIME Python Integration (Labs) extension.

Advanced

A further *Advanced* option is also available to set up the options of the pre-launched Python processes. In the background, KNIME Analytics Platform initializes and maintains a pool of Python processes that can be used by individual Python nodes, reducing the start-up cost when executing any Python node. Here, you can set up the pool size in the field *Maximum number of provisioned processes*, and the duration in minutes before recycling idle processes in the pool in the field *Expiration duration of each process (in minutes)*.



Prelaunched Python processes

In the background, KNIME initializes and maintains a pool of Python processes for use by individual Python nodes. This reduces the startup cost when executing any Python nodes. The pool size and the duration (in minutes) before recycling idle processes in the pool can be modified from their recommended defaults below.

Maximum number of provisioned processes 3

Expiration duration of each process (in minutes) 5

Troubleshooting

Mac Matplotlib

On Mac, there may be issues with the `matplotlib` package. The following error:

```
libc++abi.dylib: terminating with uncaught exception of type NSException
```

can be resolved by executing the following commands:

```
mkdir ~/.matplotlib
echo "backend: TkAgg" > ~/.matplotlib/matplotlibrc
```

Python version support

The KNIME Python Integration supports both Python 2 (2.7) and 3 (3.6 - 3.9), while the newly released KNIME Python Integration (Labs) supports Python versions 3.6 - 3.9.

MDF Reader

Similar to the KNIME Deep Learning Integration, the MDF Reader node requires certain Python packages to be installed in the **Python 3** environment. These will be installed automatically if you set up your Python environment via the Conda option on the Python Preferences page (see here). Of course, you can manually install the required packages as well:

```
numpy
libiconv
asammdf=5.19.14
```

Using the Python Scripting nodes

Overview of the nodes

The KNIME Python Integration provides a wide array of nodes. Once the extension has been installed and configured, you are able to find the available nodes in the *Node Repository* area of KNIME Analytics Platform by navigating to *Scripting* → *Python*, or simply by entering **Python** in the search field.

Additionally, all the nodes included in the KNIME Python Integration can be found on the KNIME Hub, complete with detailed descriptions of their functionality, inputs and outputs, configuration dialog, and much more. In the *Related workflows & nodes* section of the KNIME Hub page for each node, you are able to see a list of published workflows that use this particular Python node. You can easily download and explore published nodes, workflows, and components locally by dragging & dropping the special icon into the corresponding area of KNIME Analytics Platform.

Here we present an overview of the nodes available in the KNIME Python Integration.

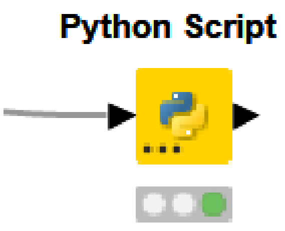
All nodes described in this section are designed to execute Python scripts in a local Python environment of your choice, support Python 2 and 3, and allow to import Jupyter Notebooks as Python modules via the `knime_jupyter` module available in the corresponding node's Python workspace.

Python Source

Python Source

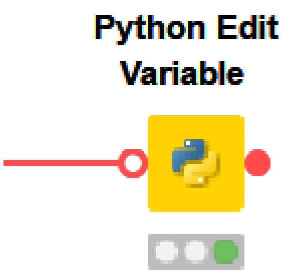
The node outputs a *KNIME table*.

Python Script



Unlike the Python Source node, this node allows for multiple input and output ports of various types, which can be dynamically added or removed via the three dots button located in the bottom left corner of the node. The default input/output ports use *KNIME data tables*, with additional options being *pickled objects* for input and output, and *images* for output.

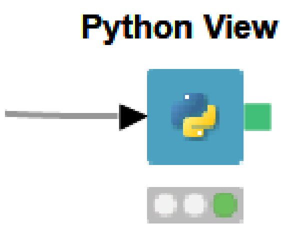
Python Edit Variable



As input and output, the node takes flow variables. The Python script can edit flow variables that have been provided as input, as well as create new flow variables.

Technically, all Python nodes available in the KNIME Python Integration are able to edit and create flow variables, which can then be propagated using the *hidden* flow variable output port that every node has. These ports can be revealed by right-clicking the node in your KNIME Analytics Platform *Workflow editor*, and selecting *Show Flow Variable Ports*. Moreover, flow variables are automatically propagated to downstream nodes via other types of connections as well.

Python View

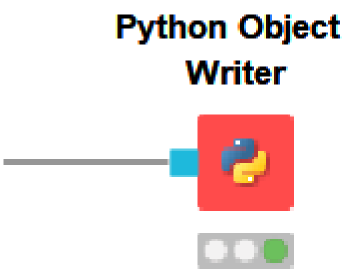


The node outputs an *image*.

Python Object Reader

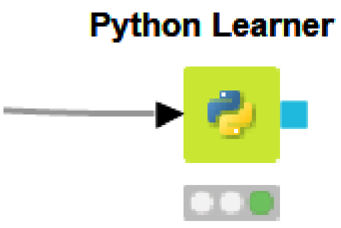
The Python script inside the node reads a Python *object*, which can be a pickle or any datatype that can be pickled. The output of the node can then be provided as input to the Python Script node, for example.

Python Object Writer



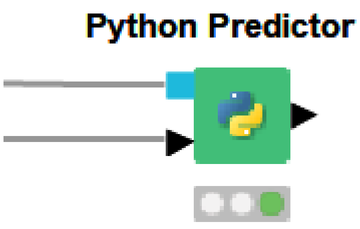
The node expects an *object* as input, which can contain any datatype that can be pickled.

Python Learner



Given a KNIME data table as input, the node is designed to output a trained model as an *object*, which can be of any datatype that can be pickled.

Python Predictor



Given a trained model *object* and a KNIME data table as input, the node is designed to produce inference by applying the model to the data inside the Python script. The node outputs a KNIME data table.

Node configuration settings

Each Python node comes with a set of configuration settings specific to the KNIME Python

Integration nodes:

- **Script**

  The code editor section of the node configuration dialog. The code for your Python script goes here. In dedicated areas of this dialog, you can see the input and output data, the available flow variables, as well as the variables of the current Python workspace.

  > In the **Script** section of the configuration dialog, you have two options of executing your Python script without leaving the dialog itself: *Execute script*, which is useful if you want to quickly check if your code is working as intended; and *Execute selected lines*, which allows you to run specific lines inside your script. This is convenient for debugging purposes, and, additionally, enables exploratory programming when, for instance, working with imported Jupyter Notebooks as described in this section of the guide.

  > Additionally, the code editor in the **Script** section provides code autocompletion. By typing a `.` and pressing `ctrl-space` (or `command-space` on Mac), you can view the available properties and methods for a given variable, or the classes and functions provided by a module. For this functionality to work, make sure that the **Jedi** package is installed in your Python environment. If your Python environment was automatically created on the Python Preferences page as described in this section, it will already contain **Jedi**.

- **Options**

  Here you can configure certain aspects of the behavior of the Python node, such as limiting the number of rows from the input table (if applicable) available to the Python script when executing inside the configuration dialog, or handling missing values in your data.

- **Executable Selection**

  This section allows you to choose which version of Python to use in this particular node (this option defaults to the version of Python selected in Python Preferences as seen in the Configure and manage Python environments section). Here you can also make use of the Conda Environment Propagation flow variable as described in the Configure and export Python environments section of this guide.

- **Templates**

For each Python node, this section of the configuration dialog will contain a number of templates that demonstrate the basic usage of the node. You can copy sections of the provided Python code into your script, or use the entire template as a starting point. Additionally, you can create custom templates using your Python code from the *Script* tab of the configuration dialog.

- **Flow Variables**

  This section of the configuration dialog allows you to automate certain other aspects of the node's configuration, including some of the options mentioned above.

You can find more details about the configuration options for each node in the KNIME Python Integration on the corresponding KNIME Hub page for the node, or in the *Description* area of KNIME Analytics Platform after selecting the node in the *Workflow editor*.

Examples of usage

You can find a comprehensive and diverse list of workflows using the nodes available in the KNIME Python Integration by searching for **Python** on the KNIME Hub and navigating to the *Workflows* section of the search results.

Preferences page

By going to *Preferences* in KNIME Analytics Platform, and then navigating to *KNIME → Python*, you can find additional settings that we described in detail in the Configuring the KNIME Python Integration section.

Using the Python Script (Labs) node

Introduction

With the v4.5 release of KNIME Analytics Platform, we are introducing the Python Script (Labs) node. Currently available as part of the KNIME Python Integration (Labs) extension (which you can install following the steps described here), this node provides a glimpse into the future of Python in KNIME Analytics Platform.

Feature highlight

- Improved performance
- A new API via the `knime_io` module
- Support conversion to both *Pandas DataFrames* and *PyArrow Tables*
- Support for arbitrarily large datasets using *batches*.

Thanks to the new backend powered by Apache Arrow, the new Python Script (Labs) node provides a significant boost in processing performance and data transfers between Python and KNIME Analytics Platform.

For a complete documentation of the new API please refer to KNIME Python Script (Labs) API documentation.

To achieve biggest possible performance gains, we recommend configuring your workflows to use Columnar Backend which is included in KNIME Analytics Platform as of the v4.5 release. Right-click the appropriate workflow in KNIME Explorer, select *Configure...*, then choose the **Columnar Backend** option under *Selected Table Backend*.

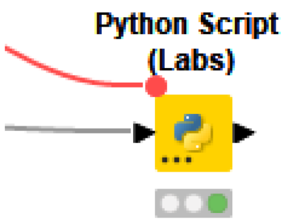
Another notable change is the introduction of the `knime_io` module. This module provides a new, more Pythonic way of accessing and working with data inside your Python scripts.

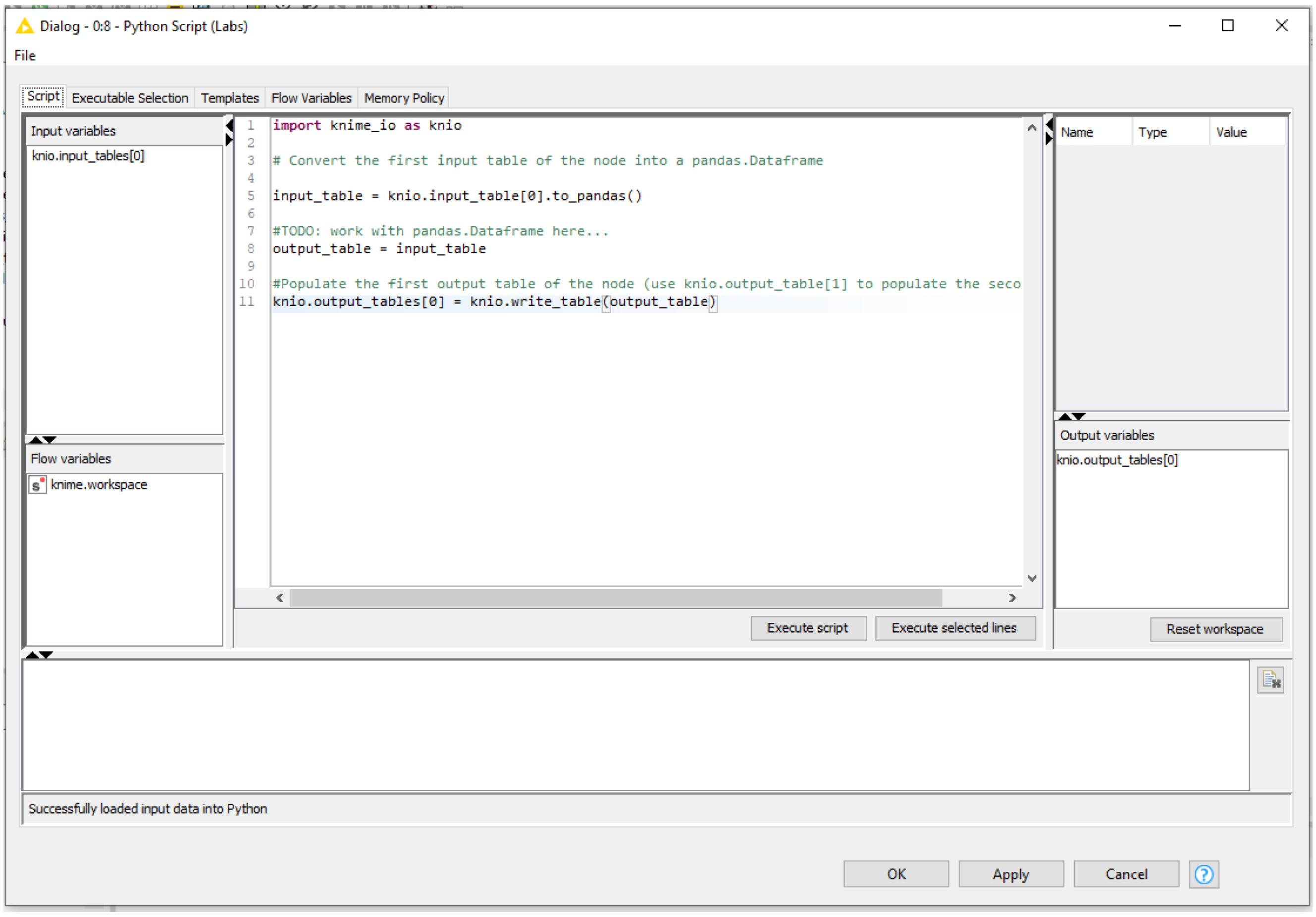
An exciting new functionality that comes with the `knime_io` module is the ability to process data in **batches**. Whereas previously the size of the input data was limited by the amount of RAM available on the machine, the Python Script (Labs) node can process arbitrarily large amounts of data by accessing it in batches via the `.batches()` method of the input table.

We will demonstrate the use of `knime_io` in the Examples section below.

Configuration

Similar to the Python nodes found in the non-Labs KNIME Python Integration, the Python Script (Labs) node contains several sections in the configuration dialog.




Dialog - 0:8 - Python Script (Labs)

File

Script | Executable Selection | Templates | Flow Variables | Memory Policy

Input variables: knio.input_tables[0]

Flow variables: knime.workspace

```
import knime_io as knio

# Convert the first input table of the node into a pandas.Dataframe

input_table = knio.input_table[0].to_pandas()

#TODO: work with pandas.Dataframe here...
output_table = input_table

#Populate the first output table of the node (use knio.output_table[1] to populate the seco
knio.output_tables[0] = knio.write_table(output_table)
```

Name	Type	Value

Output variables: knio.output_tables[0]

Execute script | Execute selected lines | Reset workspace

Successfully loaded input data into Python

OK | Apply | Cancel

Script

As the figure above demonstrates, functionality of the input, output, and flow variable panes is condensed in the `knime_io` module. We demonstrate the new way of accessing data in the Examples of usage section below.

Adding and removing ports

Similar to the non-Labs Python Script node, the new node allows adding and removing input and output ports by clicking the *three dot button* located in the bottom left corner of the node. The default input/output ports use *KNIME data tables*, with additional options being *pickled objects* for input and output, and *images* for output.

Other

For an overview of the other sections of the configuration dialog, please refer to this section of the guide.

Examples of usage

When you create a new instance of the Python Script (Labs) node, the code editor will already contain starter code, in which we import the `knime_io` module.

Accessing data

As mentioned before, `knime_io` provides a new way of accessing the data coming into the node. Namely, the input and output tables and objects can now be accessed from respective Python lists

- `knime_io.input_tables[i]` and `knime_io.output_tables[i]`
- `knime_io.input_objects[i]` and `knime_io.output_objects[i]`,
- `knime_io.output_images[i]` to output images, which must be either a string describing an SVG image or a byte array encoding a PNG image,

  where `i` is the index of the corresponding table/object/image (`0` for the first input/output port, `1` for the second input/output port, and so on).

Flow variables can be accessed from the dictionary:

- `knime_io.flow_variables['name_of_flow_variable']`.

Converting input tables to Pandas DataFrames and PyArrow Tables

The `knime_io` module provides a simple way of accessing the input data as a Pandas DataFrame or PyArrow Table. This can prove quite useful since the two data representations and corresponding libraries provide a different set of tools that might be applicable to different use-cases.

- Converting the first input table to a Pandas DataFrame using the `to_pandas()` method:

  ```
  input_df = knime_io.input_tables[0].to_pandas()
  ```

- Converting the first input table to a PyArrow Table using the `to_pyarrow()` method:

  ```
  input_table = knime_io.input_tables[0].to_pyarrow()
  ```

Working with batches

The Python Script (Labs) node, together with the `knime_io` module, allows efficiently processing larger-than-RAM data tables by utilising batching.

1. First, you need to initialise an instance of a table to which the batches will be written after being processed:

```
processed_table = knime_io.batch_write_table()
```

2. Calling the `batches()` method on an input table returns an iterable, items of which are batches of the input table that can be accessed via a `for` loop:

```
processed_table = knime_io.batch_write_table()
for batch in knime_io.input_tables[0].batches():
```

3. Inside the `for` loop, the batch can be converted to a Pandas DataFrame or a PyArrow Table using the methods `to_pandas()` and `to_pyarrow()` mentioned above:

```
processed_table = knime_io.batch_write_table()
for batch in knime_io.input_tables[0].batches():
    input_batch = batch.to_pandas()
```

4. At the end of each iteration of the loop, the batch should be appended to the table initialised in 1:

```
processed_table = knime_io.batch_write_table()
for batch in knime_io.input_tables[0].batches():
    input_batch = batch.to_pandas()
    # process the batch
    processed_table.append(input_batch)
```

The **Script** section of the configuration dialog for the Python Script (Labs) node provides code autocompletion. By typing a `.` and pressing `ctrl-space` (or `command-space` on Mac), you can view the available properties and methods for a given variable, or the classes and functions provided by a module. For instance, by typing `batch.`, you can see that it contains properties such as `num_rows` and `column_names`. For this functionality to work, make sure that the **Jedi** package is installed in your Python environment. If your Python environment was automatically created on the Python Preferences page as described in this section, it will already contain **Jedi**.

Note that the **Templates** section of the configuration dialog for the node provides starter code for the use-cases described above.

Porting old Python scripts

Adapting your old Python scripts to work with the new Python Script (Labs) node is as easy as adding the following to your code:

```
import knime_io
input_table_1 = knime_io.input_tables[0].to_pandas()

# the old script goes here

knime_io.output_tables[0] = knime_io.write_table(output_table_1)
```

Note that the numbering of inputs and outputs in the Python Script (Labs) node is 0-based - keep that in mind when porting your scripts from the other Python nodes, which have a 1-based numbering scheme (e.g. `knime_io.input_tables[0]` in the Python Script (Labs) node corresponds to `input_table_1` in the other Python nodes).

Further examples

You can find an example of the usage of the Python Script (Labs) node on KNIME Hub.

Known limitations

- The Python Script (Labs) node only supports Python versions 3.6 - 3.9.
- Extension data types like KNIME Image Processing images or RDKit molecules are not yet supported.
- The new API described in this section is part of KNIME Labs, and is currently under active development. This means that features might change with future releases.

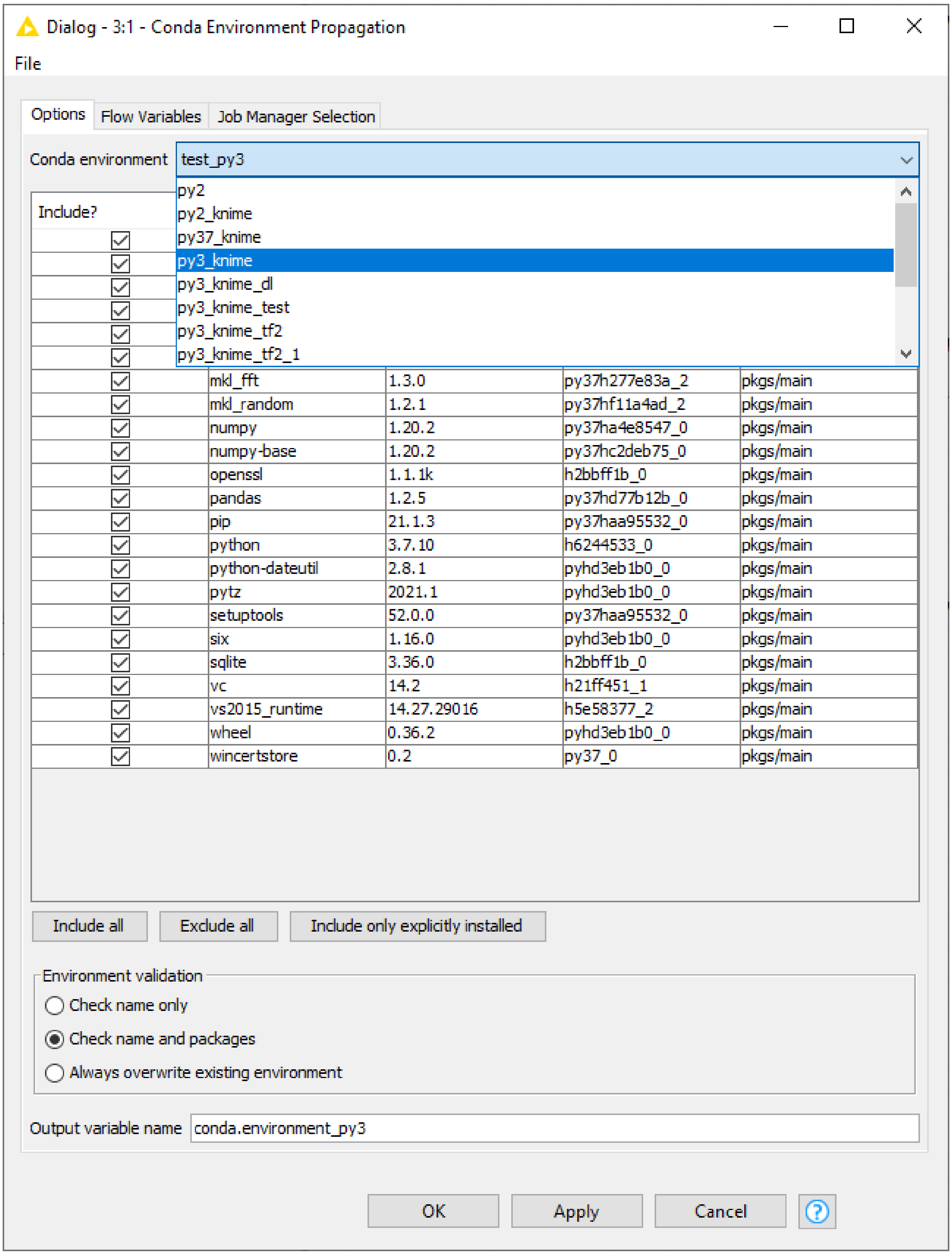
Configure and export Python environments

Besides setting up Python for your entire KNIME workspace via the Preferences page, you can also use the Conda Environment Propagation node to configure custom Python environments and then propagate them to downstream Python nodes. This node also allows you to bundle these environments together with your workflows, making it easy for other people (and other machines) to replicate the exact same environment that the workflow is meant to be executed in. This makes workflows containing Python nodes significantly more portable and less error-prone.

Configure the Python environment with Conda Environment Propagation node

To be able to make use of the Conda Environment Propagation node, you need to follow these steps:

1. On your local machine, you should have Conda set up and configured in the Preferences of the KNIME Python Integration as described in the Conda environments section
2. Open the node configuration dialog and select the Conda environment you want to propagate and the packages to include in the environment in case it will be recreated on a different machine



Dialog - 3:1 - Conda Environment Propagation

File

Options | Flow Variables | Job Manager Selection

Conda environment: test_py3

- py2
- py2_knime
- py37_knime
- py3_knime
- py3_knime_dl
- py3_knime_test
- py3_knime_tf2
- py3_knime_tf2_1

Include?				
✓				
✓				
✓				
✓				
✓				
✓				
✓	mkl_fft	1.3.0	py37h277e83a_2	pkgs/main
✓	mkl_random	1.2.1	py37hf11a4ad_2	pkgs/main
✓	numpy	1.20.2	py37ha4e8547_0	pkgs/main
✓	numpy-base	1.20.2	py37hc2deb75_0	pkgs/main
✓	openssl	1.1.1k	h2bbff1b_0	pkgs/main
✓	pandas	1.2.5	py37hd77b12b_0	pkgs/main
✓	pip	21.1.3	py37haa95532_0	pkgs/main
✓	python	3.7.10	h6244533_0	pkgs/main
✓	python-dateutil	2.8.1	pyhd3eb1b0_0	pkgs/main
✓	pytz	2021.1	pyhd3eb1b0_0	pkgs/main
✓	setuptools	52.0.0	py37haa95532_0	pkgs/main
✓	six	1.16.0	pyhd3eb1b0_0	pkgs/main
✓	sqlite	3.36.0	h2bbff1b_0	pkgs/main
✓	vc	14.2	h21ff451_1	pkgs/main
✓	vs2015_runtime	14.27.29016	h5e58377_2	pkgs/main
✓	wheel	0.36.2	pyhd3eb1b0_0	pkgs/main
✓	wincertstore	0.2	py37_0	pkgs/main

Include all | Exclude all | Include only explicitly installed

Environment validation

- ○ Check name only
- ● Check name and packages
- ○ Always overwrite existing environment

Output variable name: conda.environment_py3

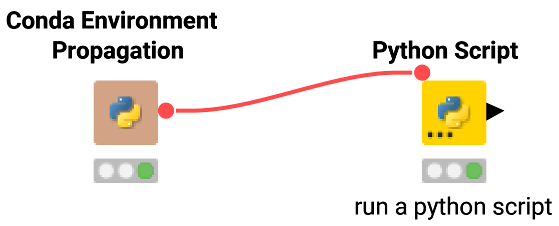
OK | Apply | Cancel

3. The Conda Environment Propagation node outputs a flow variable which contains the necessary information about the Python environment (i.e. the name of the environment and the respective installed packages and versions). The flow variable has `conda.environment` as the default name, but you can specify a custom name. This way you can avoid name collisions that may occur when employing multiple Conda

Environment Propagation nodes in a single workflow.

In order for any Python node in the workflow to use the environment you just created, you need to:

1. Connect the flow variable output port of Conda Environment Propagation node to the input flow variable port of a Python node

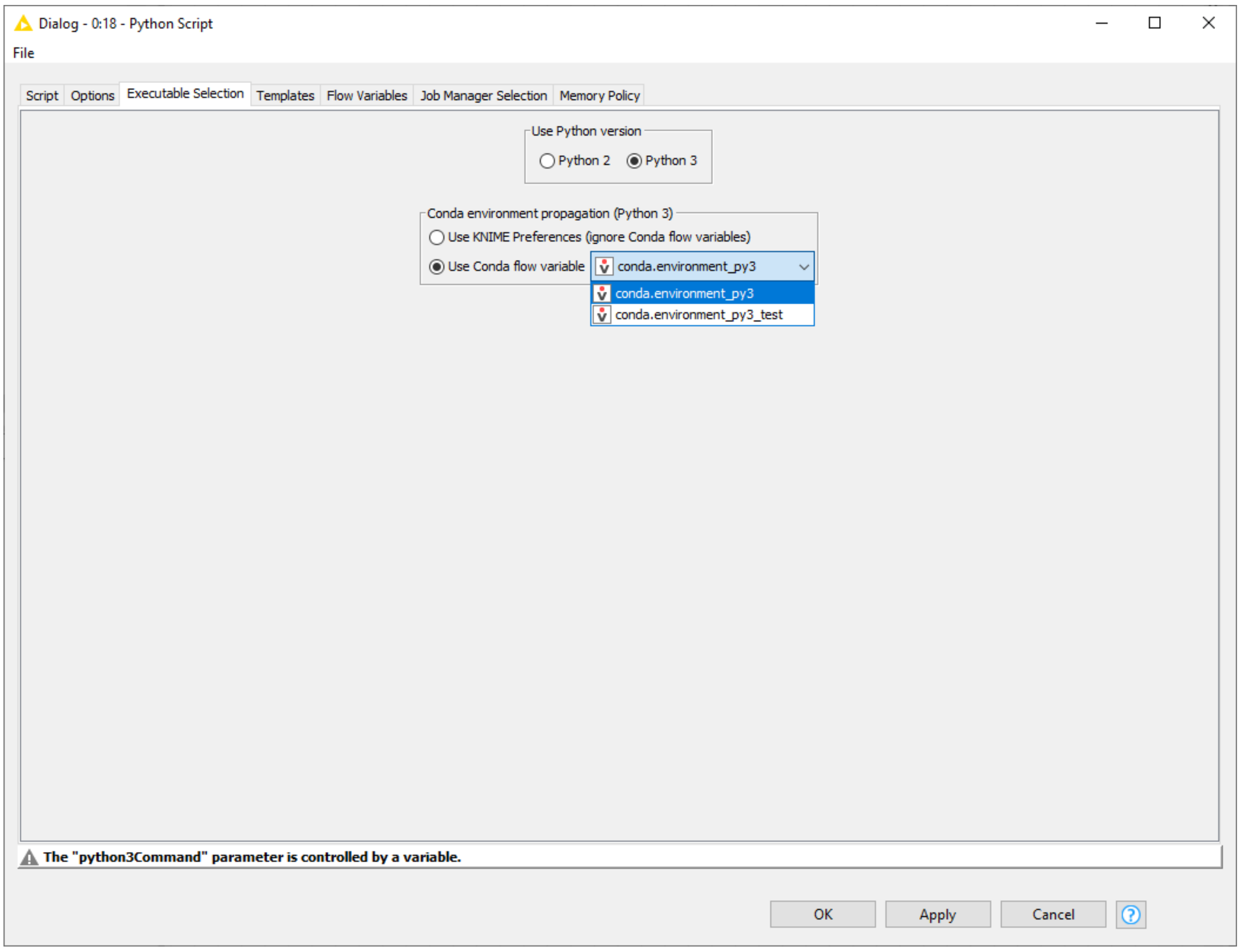


Please note that, since flow variables are propagated also through connections that are not flow variable connections, the flow variable propagating the Conda environment you created with the Conda Environment Propagation node will also be available for all downstream nodes.

2. Successively open the configuration dialogue of the Python nodes in the workflow that you want to make portable, go to the *Executable Selection* tab, and select:

   a. The Python version to be used by the current node

   b. Whether you want to use the Conda flow variable and then select the name of the Conda flow variable you want to use, or if you want the node to use the Python environment selected in the KNIME Preferences, which is the default behaviour.



Use Python version
Python 2 [✓ Python 3]

Conda environment propagation (Python 3)
Use KNIME Preferences (ignore Conda flow variables)
[✓ Use Conda flow variable] conda.environment_py3
conda.environment_py3
conda.environment_py3_test

The "python3Command" parameter is controlled by a variable.

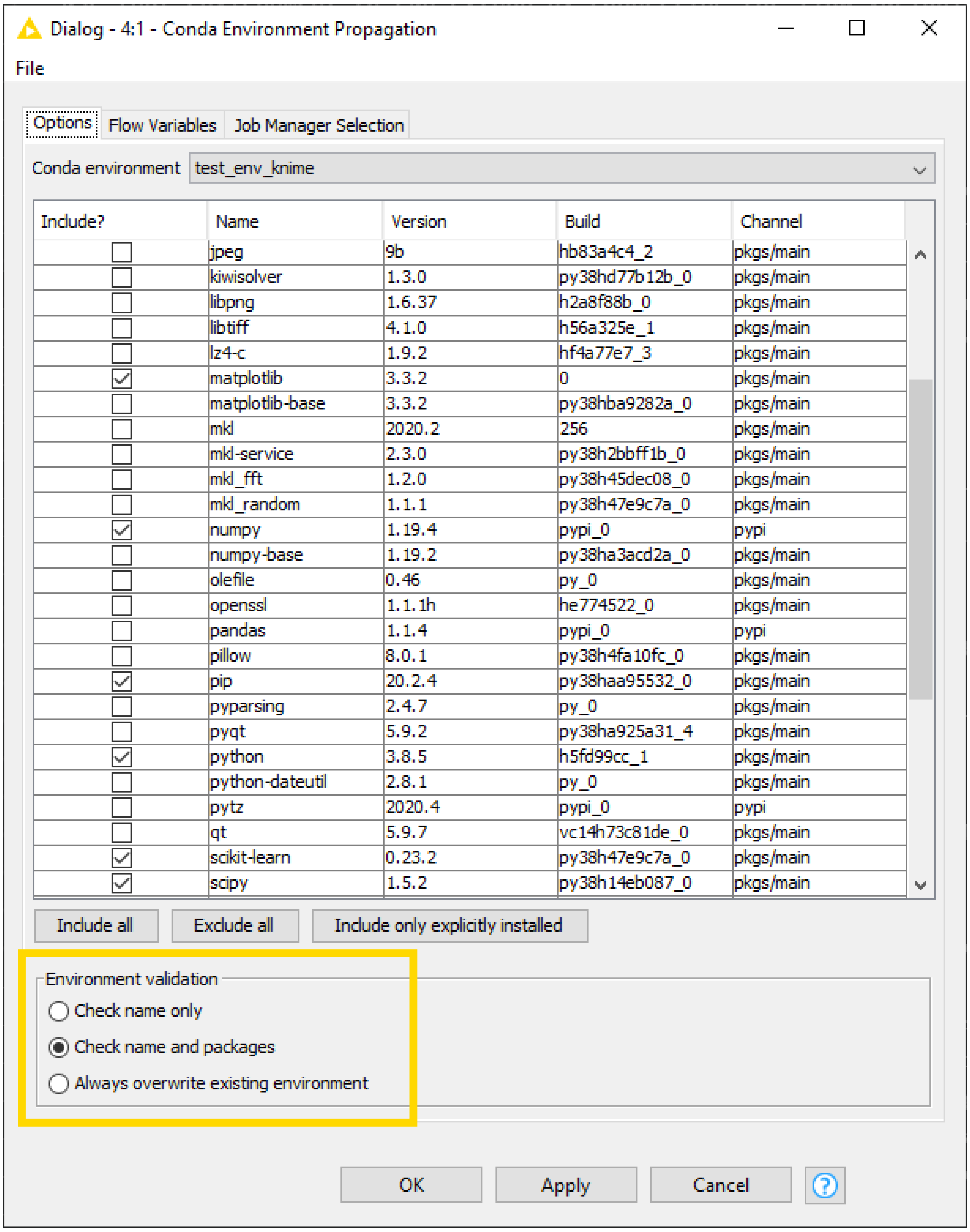
Export a Python environment with a workflow

Once you configured the Conda Environment Propagation node and set up the desired workflow, you might want to run this workflow on a target machine, for example a KNIME Server instance.

1. Deploy the workflow by uploading it to the KNIME Server, sharing it via the KNIME Hub, or exporting it. Make sure that the Conda Environment Propagation node is reset before or during the deployment process.
2. On the target machine, Conda must also be set up and configured in the Preferences of the KNIME Python Integration. If the target machine runs a KNIME Server, you may need to contact your server administrator and/or refer to the Server Administration Guide in order to do this.
3. During execution (on either machine), the node will check whether a local Conda environment exists that matches its configured environment. When configuring the node, you can choose which modality will be used for the Conda environment validation on the target machine. *Check name only* will only check for the existence of an environment with the same name as the original one, *Check name and packages* will

check both name and requested packages, while *Always overwrite existing environment* will disregard the existence of an equal environment on the target machine and will recreate it.

Depending on the above configuration, the execution time of the node will vary. For instance, a simple Conda environment name check will be much faster than a name and package check, which, in turn, will be faster than a full environment recreation process.



Dialog - 4:1 - Conda Environment Propagation

Conda environment: test_env_knime

Include?	Name	Version	Build	Channel
☐	jpeg	9b	hb83a4c4_2	pkgs/main
☐	kiwisolver	1.3.0	py38hd77b12b_0	pkgs/main
☐	libpng	1.6.37	h2a8f88b_0	pkgs/main
☐	libtiff	4.1.0	h56a325e_1	pkgs/main
☐	lz4-c	1.9.2	hf4a77e7_3	pkgs/main
✓	matplotlib	3.3.2	0	pkgs/main
☐	matplotlib-base	3.3.2	py38hba9282a_0	pkgs/main
☐	mkl	2020.2	256	pkgs/main
☐	mkl-service	2.3.0	py38h2bbff1b_0	pkgs/main
☐	mkl_fft	1.2.0	py38h45dec08_0	pkgs/main
☐	mkl_random	1.1.1	py38h47e9c7a_0	pkgs/main
✓	numpy	1.19.4	pypi_0	pypi
☐	numpy-base	1.19.2	py38ha3acd2a_0	pkgs/main
☐	olefile	0.46	py_0	pkgs/main
☐	openssl	1.1.1h	he774522_0	pkgs/main
☐	pandas	1.1.4	pypi_0	pypi
☐	pillow	8.0.1	py38h4fa10fc_0	pkgs/main
✓	pip	20.2.4	py38haa95532_0	pkgs/main
☐	pyparsing	2.4.7	py_0	pkgs/main
☐	pyqt	5.9.2	py38ha925a31_4	pkgs/main
✓	python	3.8.5	h5fd99cc_1	pkgs/main
☐	python-dateutil	2.8.1	py_0	pkgs/main
☐	pytz	2020.4	pypi_0	pypi
☐	qt	5.9.7	vc14h73c81de_0	pkgs/main
✓	scikit-learn	0.23.2	py38h47e9c7a_0	pkgs/main
✓	scipy	1.5.2	py38h14eb087_0	pkgs/main

Environment validation
- ○ Check name only
- ● Check name and packages
- ○ Always overwrite existing environment

Please be aware that exporting Python environments between systems that run different Operating Systems might cause some libraries to conflict.

Manual configuration of Python environments per node

In case you do not want to use the Conda Environment Propagation node's functionality, you can also configure individual nodes manually to use specific Python environments. This is done via flow variables `python2Command` and `python3Command` that each Python scripting node offers under the *Flow Variables* tab in its configuration dialog. Both variables accept the path to a Python start script like in the Manual case described above. Which of the two flow variables is respected depends on whether a node is using Python 2 or Python 3. This can either be configured via option *Use Python Version* under the *Executable Selection* tab in the node's configuration dialog or via flow variable `pythonVersionOption` which accepts either *python2* or *python3* as value.

Load Jupyter notebooks from KNIME

Existing Jupyter notebooks can be accessed within Python Scripting nodes using the `knime_jupyter` Python module (`knime_jupyter` will be imported automatically). Notebooks can be opened via the function `knime_jupyter.load_notebook`, which returns a standard Python module. The `load_notebook` function needs the path to the folder that contains the notebook file and the filename of the notebook as arguments. After a notebook has been loaded, you can call functions that are defined in the code cells of the notebook like any other function of a Python module. Furthermore, you can print the textual content of each cell of a Jupyter notebook using the function `knime_jupyter.print_notebook`. It takes the same arguments as the `load_notebook` function.

An example script for a Python Script node loading a notebook could look like this:

```
# Path to the folder containing the notebook, e.g. the folder 'data' contained
# in my workflow folder
notebook_directory = "knime://knime.workflow/data/"

# Filename of the notebook
notebook_name = "sum_table.ipynb"

# Load the notebook as a Python module
my_notebook = knime_jupyter.load_notebook(notebook_directory, notebook_name)

# Print its textual contents
knime_jupyter.print_notebook(notebook_directory, notebook_name)

# Call a function 'sum_each_row' defined in the notebook
output_table = my_notebook.sum_each_row(input_table)
```

The `load_notebook` and `print_notebook` functions have two optional arguments:

- `notebook_version`: The Jupyter notebook format major version. Sometimes the version can't be read from a notebook file. In these cases, this option allows to specify the expected version in order to avoid compatibility issues. Should be an integer.
- `only_include_tag`: Only load cells that are annotated with the given custom cell tag (since Jupyter 5.0.0). This is useful to mark cells that are intended to be used in a Python module. All other cells are excluded. This is e.g. helpful to exclude cells that do visualization or contain demo code. Should be a string.

The Python nodes support code completion similar to an IDE. Just hit `ctrl-space` (`command-space` on Mac) e.g. after `knime_jupyter.` in order to show the available methods and documentation (`knime_jupyter` refers to the imported `knime_jupyter` Python module, e.g. see script example above).

The Jupyter notebook support for the KNIME Python Integration depends on the packages `IPython`, `nbformat`, and `scipy`, which are already included if you either used the automatic Conda environment creation option in the Python Preferences, or the YAML configuration files.

You can find example workflows using the `knime_jupyter` Python module on our EXAMPLES server.

Open for Innovation KNIME

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