Open for Innovation
KNIME

KNIME Python Integration Guide

KNIME AG, Zurich, Switzerland
Version 4.7 (last updated on 2023-07-03)

Table of Contents

Introduction

This guide describes how to set up and use the KNIME Python Integration in KNIME Analytics Platform with its two nodes: Python Script node and Python View node.

In the v4.5 release of KNIME Analytics Platform, we introduced the Python Script (Labs) node, which is since the v4.7 release the current Python Script node of this guide.

The KNIME Python Integration works with Python versions 3.6 and higher, but is actively only supported for 3.9 and comes with a bundled Python environment to let you start right away. This convenience allows to use the nodes without installing, configuring or even knowing environments. The included bundled Python environment comes with these packages.

To start right away, drag and drop the extension *KNIME Python Integration* from the KNIME Hub into the workbench to install it or manually via *File* → *Install KNIME Extensions…*. Then proceed to Using the Python nodes.

The section Using the Python nodes explains how the configuration of the dialogs can be used, as well as how to work with data coming to and going out of the nodes, how to work with batches and how to use the Python Script node with scripts of older Python nodes. It also provides the use-case of using Jupyter notebooks and references further examples.

If you need packages, that are not included in the bundled environment, you need to set up your own environment. In the section Configure the Python Environment the different options to set up and change environments are explored.

The API of the Python Integration can be found at Read The Docs.

Before the v4.7 release, this extension was in labs and the KNIME Python Integration (legacy) was the current Python Integration. For anything related to the legacy nodes of the former KNIME Python Integration, please refer to the Python Integration guide of KNIME Analytics Platform v4.6. The advantages of the current Python Script node and the Python View node compared to legacy nodes are significantly improved performance and data transfer between Python processes and the KNIME Analytics Platform thanks to Apache Arrow, a bundled environment to start right away, a unified API via the knime.scripting.io module, conversion support to and from both *Pandas DataFrames* and *PyArrow Tables*, support for arbitrarily large data sets by using *batches*. If you look for Python 2 support, you will also need to use the KNIME Python Integration (legacy).

To achieve biggest possible performance gains, we recommend configuring your workflows to use Columnar Backend. Right-click a workflow in KNIME Explorer, select *Configure…*, then choose the **Columnar Backend** option under *Selected Table Backend*. Additional information about table backends can be found here.

Using the Python nodes

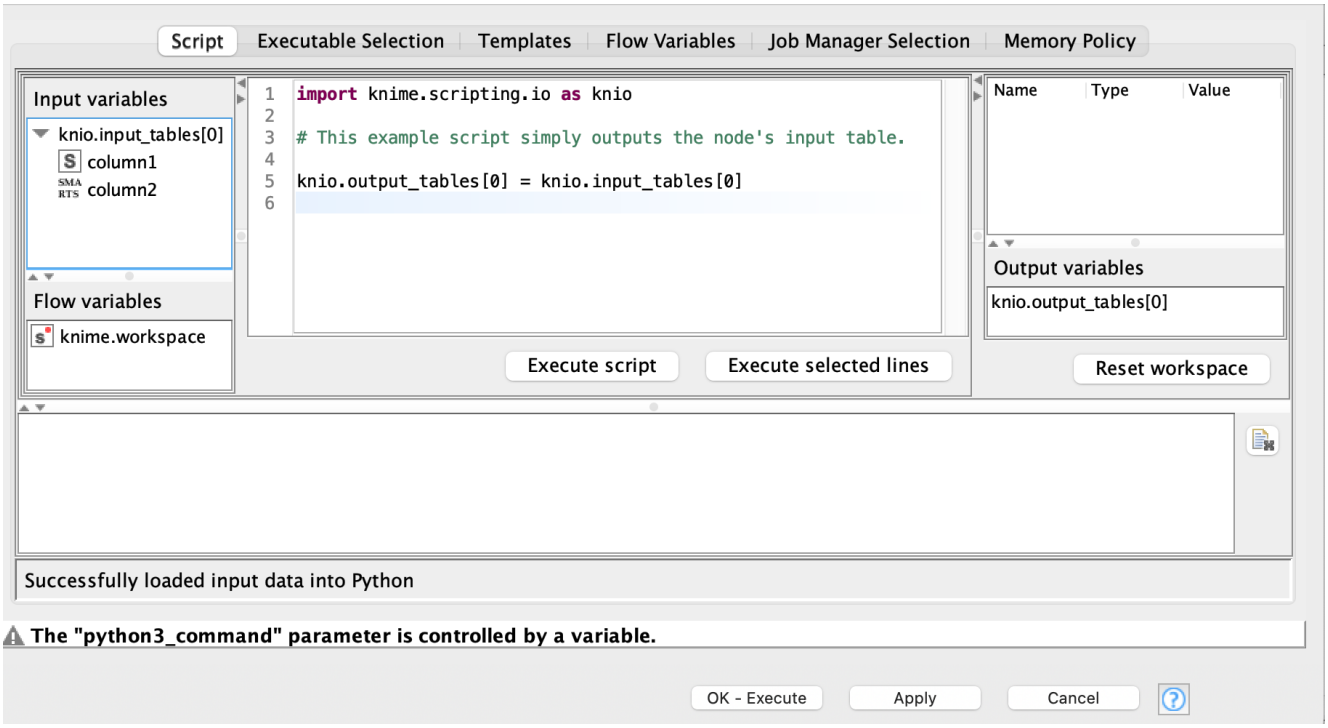
Introduction

This chapter guides through the configuration of the script dialog and the amount of ports, followed by examples of usage. These examples cover the access of input data, followed by table conversion and the usage of batches for data larger than RAM. Then it will explain how to port scripts from Python legacy nodes to this extension. After that, the additional features of the Python View node are explained. The chapter concludes with the use-case of loading and accessing Jupyter notebooks.

See the KNIME Hub for examples on using the Python nodes.

Configuration

The Python Script node and the Python View node contain several sections in the configuration dialog: *Script*, *Executable Selection*, and *Templates* among others.



- **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, the Jedi packages is preinstalled in the metapackages and the bundled environment.

- **Executable Selection**

  Here you can choose the Conda Environment Propagation flow variable as described in

the Conda propagation node section.

- **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.

Adding and removing ports Input and output ports can be added and removed via the *three dot button* located in the bottom left corner of the node. The default ports use *KNIME data tables*, with additional options being *pickled objects* for input and output, and *images* for output.

Examples of usage

When you create a new instance of the Python Script nodes, the code editor will already contain starter code, in which we `import knime.scripting.io as knio`. The content shown in the input, output, and flow variable panes can be accessed via this `knime.scripting.io` module.

Accessing data

With `import knime.scripting.io as knio`, the input and output tables and objects can be accessed from respective Python lists:

- `knio.input_tables[i]` and `knio.output_tables[i]`,
- `knio.input_objects[i]` and `knio.output_objects[i]`,
- `knio.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:

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

Converting tables to and from Pandas DataFrames and PyArrow Tables

The `knime.scripting.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 tables to and from a Pandas DataFrame:

```
df = knio.input_tables[0].to_pandas()

knio.output_tables[0] = knio.Table.from_pandas(df)
```

- Converting tables to and from a PyArrow Table:

```
table = knio.input_tables[0].to_pyarrow()

knio.output_tables[0] = knio.Table.from_pyarrow(table)
```

Working with batches

The Python nodes, together with the `knime.scripting.io` module, allow efficiently processing larger-than-RAM data tables by using batching.

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

```
processed_table = knio.BatchOutputTable.create()
```

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 = knio.BatchOutputTable.create()
for batch in knio.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 = knio.BatchOutputTable.create()
for batch in knio.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 `processed_table`:

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

The **Templates** section provides starter code for the use-cases described above.

Porting Scripts from the Python Script (Legacy) Nodes

Adapting your Python scripts from Python Script (Legacy) nodes to work with the current Python nodes is as easy as adding the following to your code:

```
import knime.scripting.io as knio
input_table_1 = knio.input_tables[0].to_pandas()

# the script from the legacy nodes goes here

knio.output_tables[0] = knio.Table.from_pandas(output_table_1)
```

Note that the numbering of inputs and outputs in the Python nodes 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. `knio.input_tables[0]` in the Python nodes corresponds to `input_table_1` in the legacy Python nodes).

Features of the Python View node

The Python View node can be used to create views using Python scripts. It has the same configurable input ports as the Python Script node and uses the same API to access the input data. However, the Python View node has no output ports except for one optional image output port.

To create a view the script must populate the variable `knio.output_view` with a return value of one of the `knio.view*` functions. It is possible to create views from all kinds of displayable objects via the convenience method `knio.view`, which tries to detect the correct format and calls the matching method of the following list of `knio.view*` functions (see API for more details):

- `knio.view` calls the appropriate of the following functions
- `knio.view_html` creates a view from a string of html content
- `knio.view_svg` creates a view from a string of svg content
- `knio.view_png` creates a view from bytes representing a png
- `knio.view_jpeg` creates a view from bytes representing a jpeg
- `knio.view_matplotlib` creates a view from the active or given matplotlib figure
- `knio.view_seaborn` creates a view from the active or given seaborn figure
- `knio.view_plotly` creates a view from a plotly figure; note that to be able to synchronize the selection between the view and other KNIME views, the custom_data of the figure traces must be set to the RowID

  Example:

  ```
  fig = px.scatter(df, x="my_x_col", y="my_y_col", color="my_label_col",
                   custom_data=[df.index]) # custom_data is set to the RowID
  node_view = view_plotly(fig)
  ```

- `knio.view_ipy_repr` creates a view from an object with an IPython `_repr_*_` function

The templates of the Python View node provide examples.

To create an output image, the optional output image port needs to be added.

The output image port is populated automatically if the view is an SVG, PNG, or JPEG image or can be converted to one. Matplotlib and seaborn figures will be converted to a PNG or SVG image depending on the format chosen in `view_matplotlib`. Plotly figures can only be converted to images if the package kaleido is installed in the environment. Objects that have an IPython `repr_svg`, `repr_png`, or `repr_jpeg` function will be converted by calling the first of these functions available. HTML documents cannot be converted to images automatically. However, it is possible to set an image representation or a function that returns an image representation when calling `view_html` (see the API).

Otherwise, the script must populate the variable `knio.output_images[0]` like in the Python

Script node.

Load Jupyter notebooks from KNIME

Existing Jupyter notebooks can be accessed within Python Scripting nodes if we `import knime.scripting.jupyter as knupyter`. Notebooks can be opened via the function `knupyter.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 `knupyter.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
import knime.scripting.jupyter as knupyter
my_notebook = knupyter.load_notebook(notebook_directory, notebook_name)

# Print its textual contents
knupyter.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 cannot be read from a notebook file. In these cases, this option allows to specify the expected version in order to avoid compatibility issue and 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 and should be a string.

The Jupyter notebook support for the KNIME Python Integration depends on the packages `IPython`, `nbformat`, and `scipy`, which are already included in the bundled environment and in the metapackage `knime-python-scripting`.

You can find example workflows using the `knime.scripting.jupyter` Python module on the KNIME Hub.

Configure the Python Environment (Advanced)

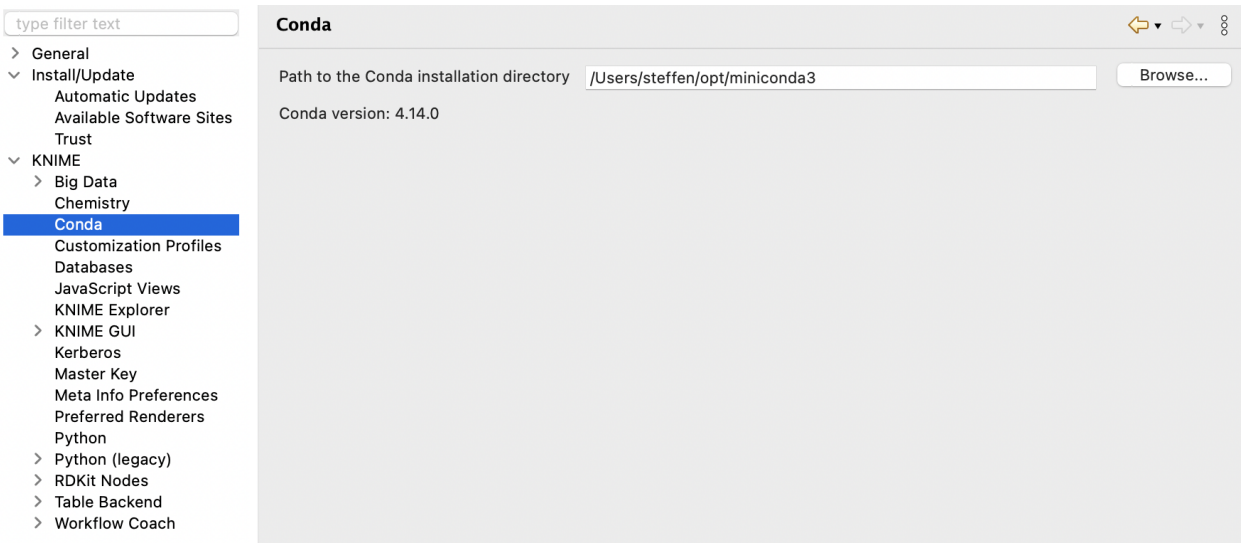
The KNIME Python Integration requires a configured Python environment. In this section we describe how to install the Python integration and how to configure its Python environment.

This section is only relevant you want to use anything different than the bundled pre-installed environment.

Besides the prerequisites, we explain possibilities for two different scopes: for the whole KNIME Analytics Platform and node-specific. The latter is handy when sharing your workflow. Lastly, the configuration for the KNIME Executor (which is used in the KNIME Business Hub) is explained in configuration example.

Prerequisites

1. Install the Python extension. Drag and drop the extension from the KNIME Hub into the workbench to install it. Or got to *File → Install KNIME Extensions* in KNIME Analytics Platform and install the *KNIME Python Integration* in the category *KNIME & Extensions*.
2. Install Conda, a package and environment manager. For instance, Miniconda, which is a minimal installation of Conda. Its initial environment, `base`, will contain a Python installation, but we recommend to create new environments for your specific use-cases. In the KNIME Analytics Platform Preferences, configure the *Path to the Conda installation directory* under *KNIME > Conda*, as shown in the following figure.



You will need to provide the path to the folder containing your installation of Conda. For Miniconda, the default installation path is

- for Windows: `C:\Users\<your-username>\miniconda3\`
- for Mac: `/Users/<your-username>/miniconda3`
- for Linux: `/home/<your-username>/miniconda3`

Once you have entered a valid path, the installed Conda version will be displayed.

> We will cover further down here how to use environments without Conda.

Configure the AP-wide environment

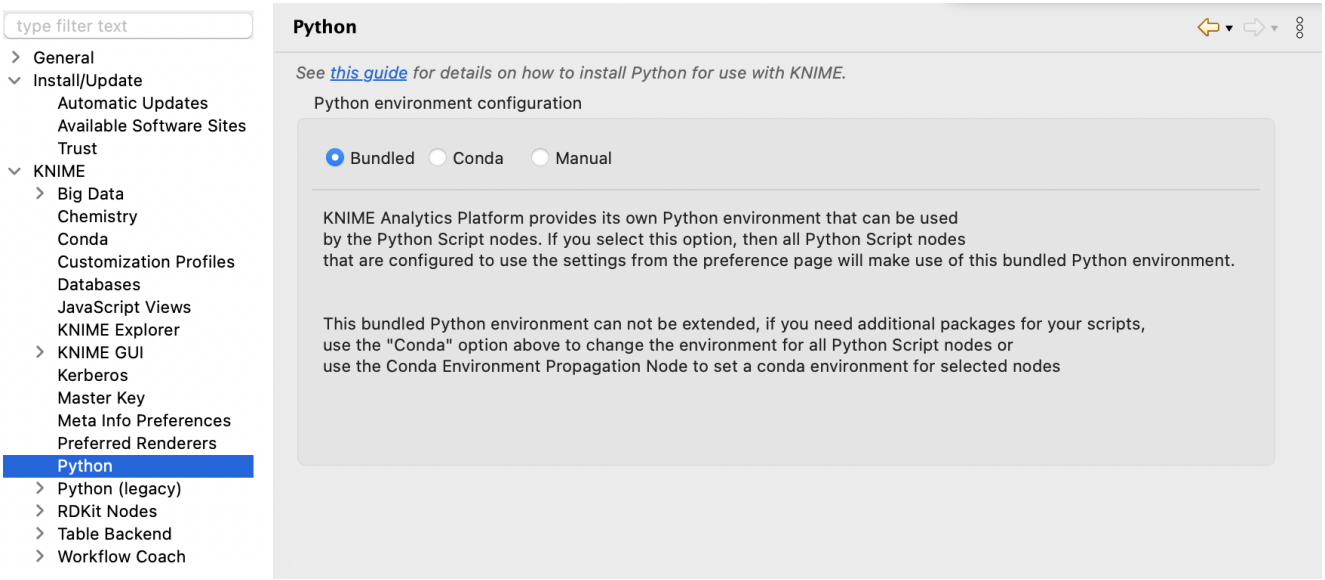
Bundled (recommended to start right away)

The KNIME Python Integration is installed with a bundled Python environment, consisting of a specific set of Python packages (i.e. Python libraries) to start right away: just open the Python Script node and start scripting.

As not everybody needs everything, this set is quite limited to allow for many scripting scenarios while keeping the bundled environment small. Thus, the list of included packages can be found in the contents of this metapackage and in the following list (with some additional dependencies):

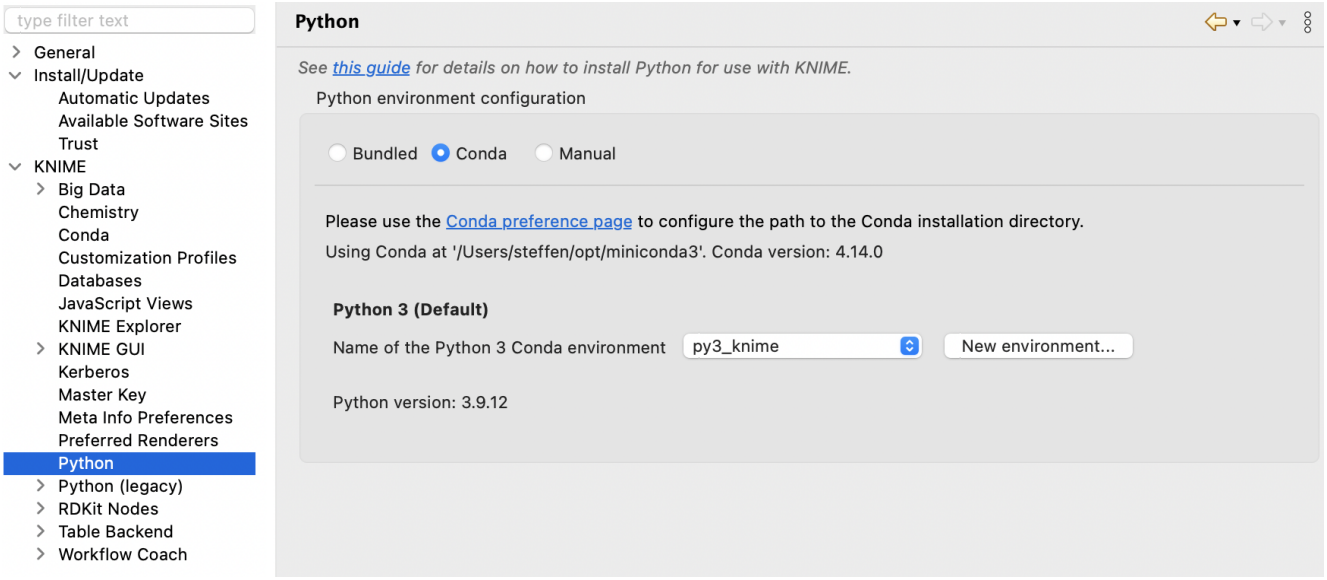
```
# Required              # Current version in the bundled environment (January 2023)
- beautifulsoup4        # 4.11.1
- cloudpickle           # 2.2.0
- ipython               # 8.8.0
- jedi>=0.18.1          # 0.18.2
- matplotlib-base       # 3.6.2
- markdown
- nbformat              # 5.7.1
- nltk                  # 3.8.1
- nomkl                 # 1.0
- numpy>=1.22           # 1.24.1
- openpyxl              # 3.0.10
- pandas                # 1.5.2
- packaging             # 21.3
- pillow                # 9.4.0
- plotly                # 5.11.0
- py4j                  # 0.10.9.7
- pyarrow>=9            # 9.0.0
- python=3.9            # 3.9.15
- python-dateutil       # 2.8.2
- pytz                  # 2022.7
- pyyaml                # 6.0
- requests              # 2.28.1
- scikit-learn          # 1.2.0
- scipy                 # 1.10.0
- seaborn               # 0.12.2
- statsmodels           # 0.13.5
```

The bundled environment is selected by default and can be reselected here:

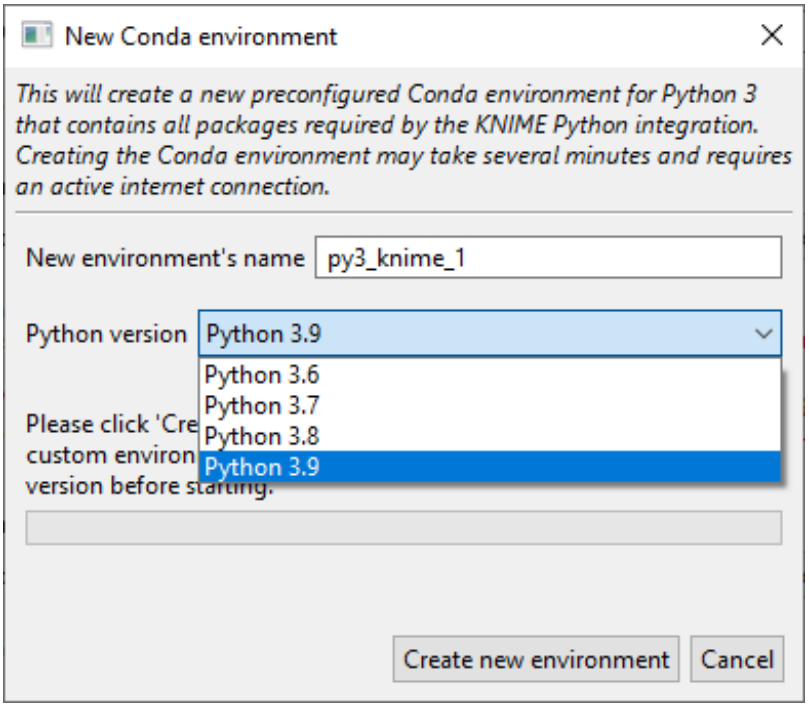


Create via Preference Page and install additional packages

Go to *KNIME > Python* and select **Conda** under *Python environment configuration*.



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:



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 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.

To further install packages, you can use any environment management tool or the command line, which is described in the following part.

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

Metapackages via terminal (recommended if additional packages are required)

If you want a Python environment with more than the packages provided by the bundled environment, you can create your environment using our metapackages. Two metapackages are important: `knime-python-base` contains the basic packages which are always needed. `knime-python-scripting` contains `knime-python-base` and installs additionally the packages used in the Python Script node. This is the set of packages which is also used in the bundled environment. Find the lists here. You can choose between different Python version (currently 3.8, 3.9 and 3.10) and select the current KNIME Analytics Platform version. See the KNIME conda channel for available versions.

Create a new environment in a terminal by adjusting and entering

```
conda create --name <ENV_NAME> -c knime -c conda-forge knime-python-scripting=4.7
python=3.9 <OTHER_PACKAGE> <OTHER_PACKAGE_WITH_VERSION_SPECIFIED>=1.2.3
```

Install additional packages into your existing environment <ENV_NAME> in the terminal by adjusting and entering

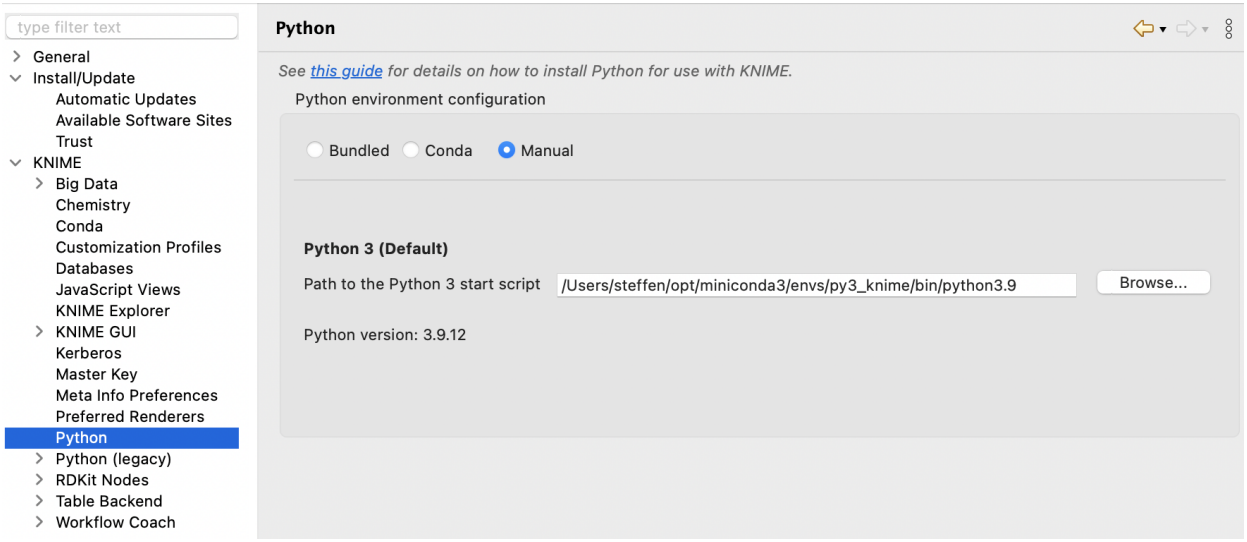
```
conda install --name <ENV_NAME> -c conda-forge <PACKAGE>
```

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

Manually specifying the Python executable/start script via the preference page

The alternative to using the Conda package manager is to manually set up the Python installation. If you choose **Manual** in the Preference page, you will have the following options:

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



2. Point KNIME Analytics Platform to a start script which activates the environment you want to use for Python 3. 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, 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 to it by specifying the path to the script on the Python Preferences page.

Configure node-specific environments

Conda Environment Propagation node

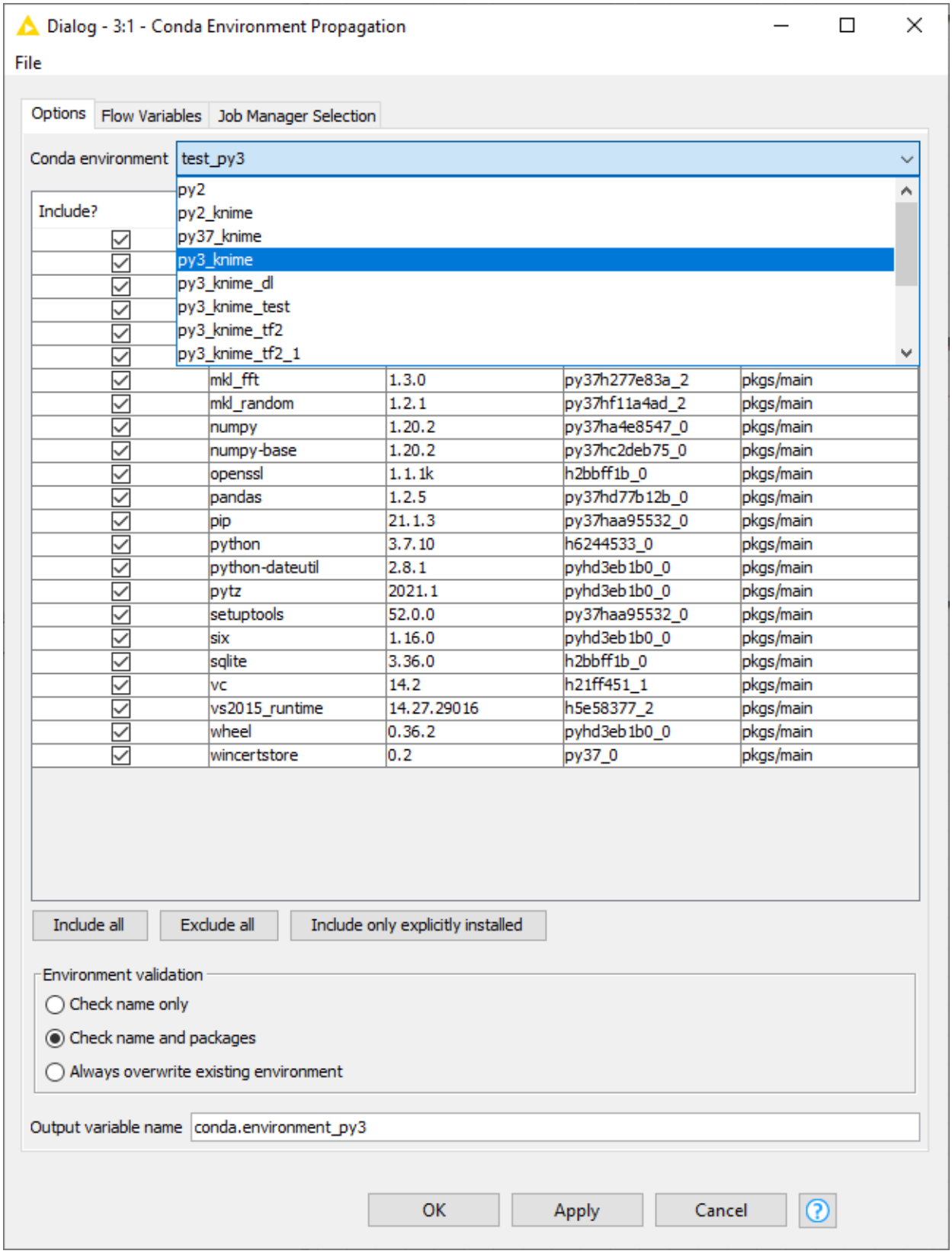
Besides setting up Python for your entire KNIME workspace via the Preferences page, you can also use the Conda Environment Propagation node to configure custom Python environments and then propagate them to downstream Python nodes. This node also allows you to bundle these environments together with your workflows, making it easy for others 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.

Setting up

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 Prerequisites 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. The packages can be selected automatically via the following buttons:

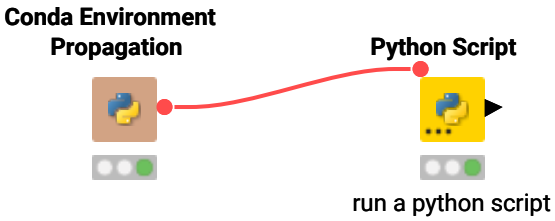
The `Include only explicitly installed` button selects only those packages that were explicitly installed into the environment by the user. This can help avoiding conflicts when using the workflow on different Operating Systems because it allows Conda to resolve the dependencies of those package for the Operating System the workflow is running on.



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.

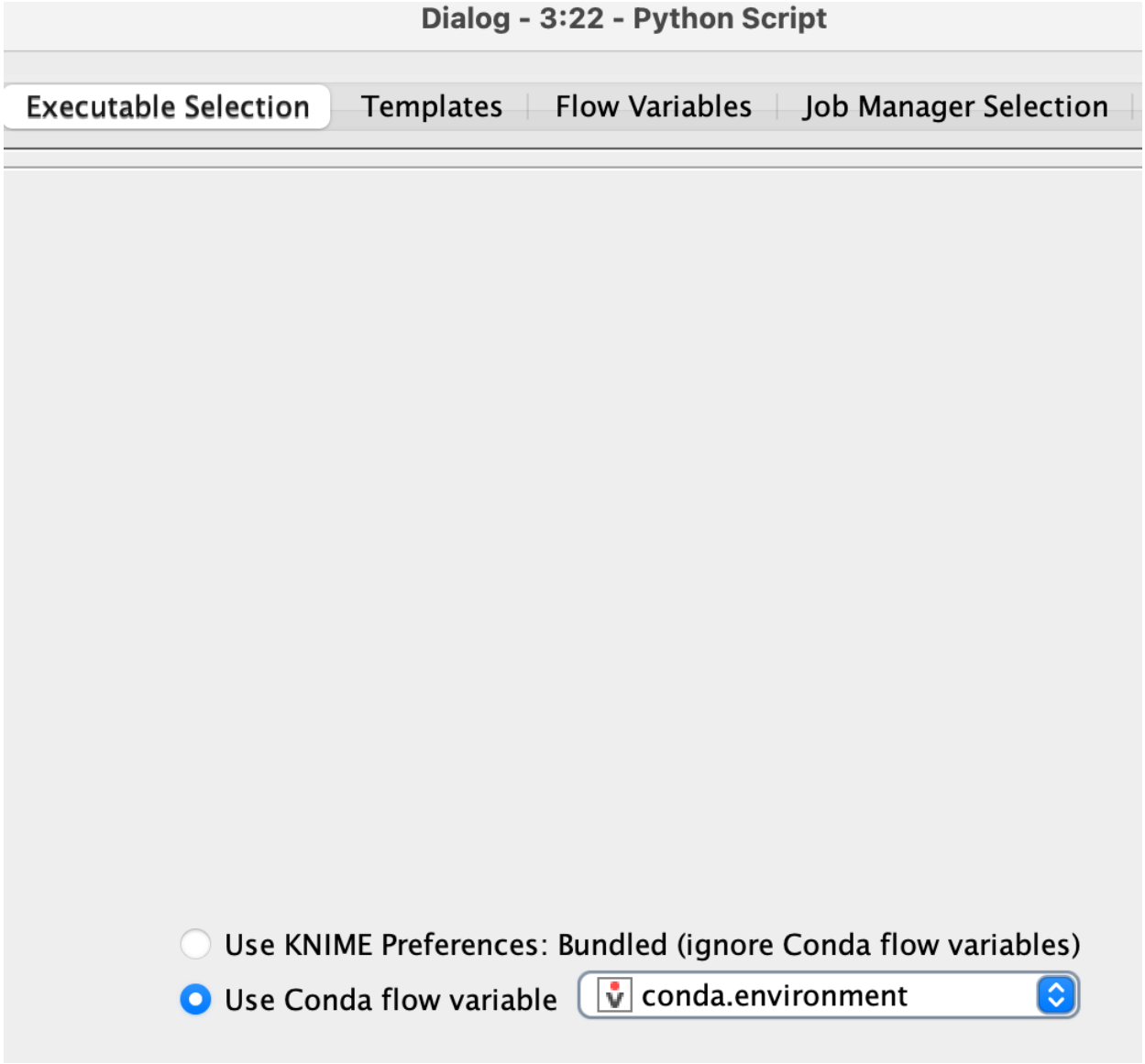
In order for any Python node in the workflow to use the environment you just created, you need to:

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 dialog of the Python nodes in the workflow that you want to make portable, go to the *Executable Selection* tab, and select which Conda flow variable you want to use.



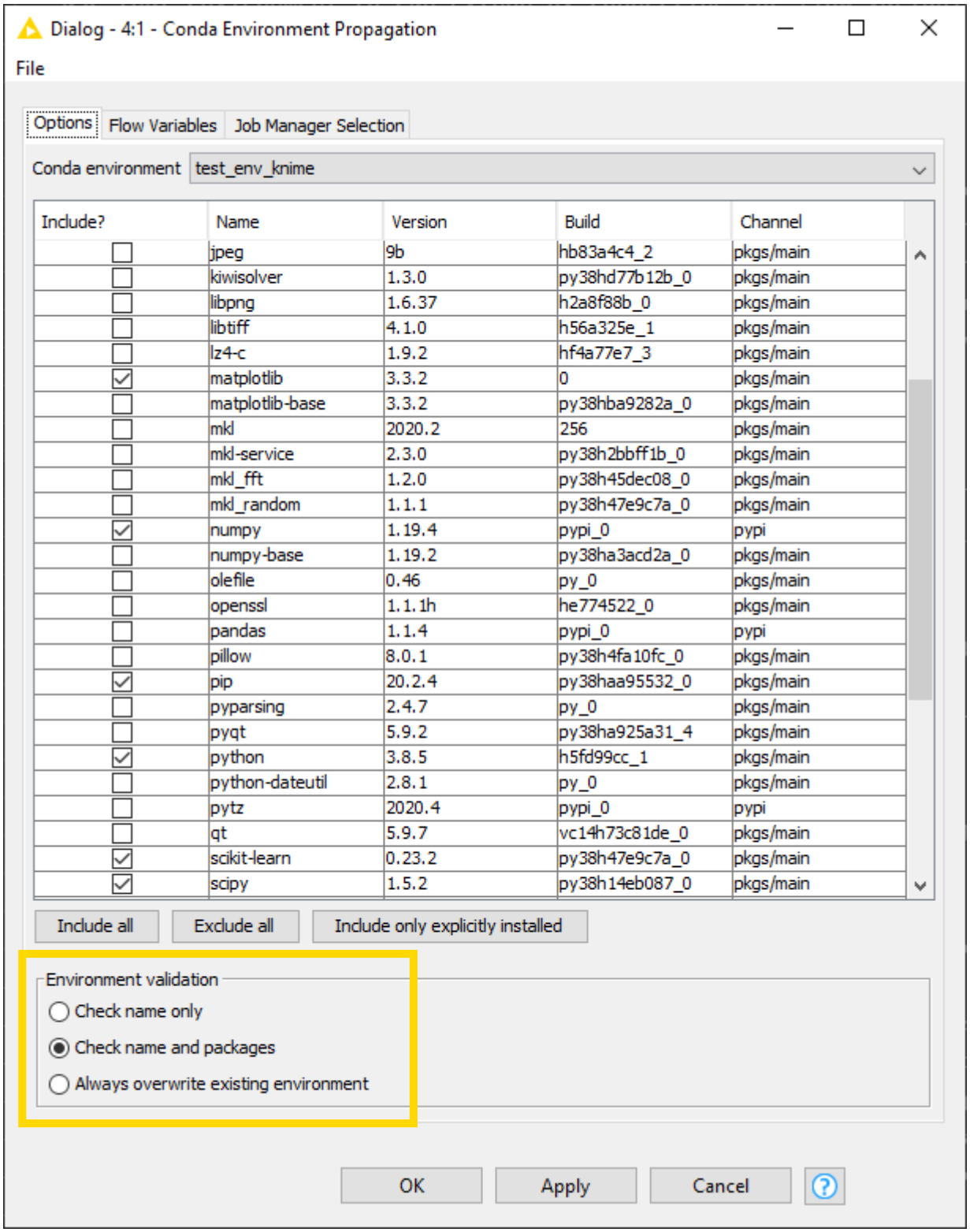
Exporting

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 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.



Exporting Python environments between systems that run different Operating Systems might cause some libraries to conflict. Please test your workflows on different Operating Systems and consider using the `Include only explicitly installed` button.

Manually specifying the Python executable/start script via flow variable

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 the flow variable `python3_command` that each Python scripting node offers under the *Flow Variables* tab in its configuration dialog. The variable accepts the path to a Python start script like in the Manual case described above.

Executor configuration

The KNIME Executor uses customization profiles, you can adapt the following parts for your convenience.

```
# A - KNIME Conda Integration - Path to Anaconda/miniconda installation directory
/instance/org.knime.conda/condaDirectoryPath=<path to conda installation dir>

# B - KNIME Python Integration - Default options for Python Integration. By default
KNIME uses the bundled environment (shipped with KNIME) if no Conda Environment
Propagation node is used.
# Line below can be set to either "bundled" (default), "conda" or "manual"
/instance/org.knime.python3.scripting.nodes/pythonEnvironmentType=bundled
/instance/org.knime.python3.scripting.nodes/bundledCondaEnvPath=org_knime_pythonscriptin
g
# Following rows are only required if "bundled" value above is replaced with "conda"
/instance/org.knime.python3.scripting.nodes/python2CondaEnvironmentDirectoryPath=<path
to default conda environment dir>
/instance/org.knime.python3.scripting.nodes/python3CondaEnvironmentDirectoryPath=<path
to default conda environment dir>
# Following rows are only required if "bundled" value above is replaced with "manual"
/instance/org.knime.python3.scripting.nodes/python2Path=<path to python2 env>
/instance/org.knime.python3.scripting.nodes/python3Path=<path to python3 env>

# C - KNIME Python Integration (Legacy) - Default options for Python Integration.
# Line below can be set to either "conda" or "manual"
/instance/org.knime.python2/pythonEnvironmentType=conda
/instance/org.knime.python2/defaultPythonOption=python3
/instance/org.knime.python2/serializerId=org.knime.python2.serde.arrow
# Following rows are only required if "conda" is set above
/instance/org.knime.python2/python2CondaEnvironmentDirectoryPath=<path to default conda
environment dir>
/instance/org.knime.python2/python3CondaEnvironmentDirectoryPath=<path to default conda
environment dir>
# Following rows are only required if "conda" value above is replaced with "manual"
/instance/org.knime.python2/python2Path=<path to python2 env>
/instance/org.knime.python2/python3Path=<path to python3 env>


# D - KNIME Deep Learning Integration
# Select either "python" or "dl" (without quotation marks) in next row. If "python" is
used, the configuration of section B above is reused. If "dl" is used, a custom config
for Deep Learning can be provided.
/instance/org.knime.dl.python/pythonConfigSelection=python
# Following rows only required if row above is set to "dl"
/instance/org.knime.dl.python/kerasCondaEnvironmentDirectoryPath=<path to default conda
environment dir>
/instance/org.knime.dl.python/librarySelection=keras
/instance/org.knime.dl.python/manualConfig=python3
/instance/org.knime.dl.python/pythonEnvironmentType=conda
/instance/org.knime.dl.python/serializerId=org.knime.python2.serde.arrow
/instance/org.knime.dl.python/tf2CondaEnvironmentDirectoryPath=<path to default conda
environment dir>
/instance/org.knime.dl.python/tf2ManualConfig=python3
```

Troubleshooting

In case you run into issues with KNIME's Python integration, here are some useful tips to help you gather more information and maybe even resolve the issue yourself. In case the issues persist and you ask for help, please include the gathered information.

Find Debug Information

Resourceful information helps in understanding issues. Relevant information can be obtained in the following ways.

Accessing the KNIME Log

The `knime.log` contains information logged during the execution of nodes. To obtain it, there are two ways:

- In the KNIME Analytics Platform: `View → Open KNIME log`
- In the file explorer: `<path-to-knime-workspace>/.metadata/knime/knime.log`

Not all logged information is required. Please restrict the information you provide to the issue. If the log file does not contain sufficient information, you can change the logging verbosity in `File → Preferences → KNIME`. You can even log the information to the console in the KNIME Analytics Program: `File → Preferences → KNIME → KNIME GUI`.

Information about the Python environment

If `conda` is used, obtain the information about the used Python environment `<python_env>` via:

1. `conda activate <python_env>`
2. `conda env export`

Information about a failed installation

If the error `An error occured while installing the items` appears when installing an extension with a bundled Python environment (the KNIME Python Integration itself and pure Python extensions), you can obtain the corresponding log files as follows. The error message contains a `<plugin_name>` like `org.knime.pythonscripting.channel.v1.bin⋯` or `sdl.harvard.geospatial.channel.bin⋯`

1. Windows/Linux: go to the folder of the KNIME Analytics Platform installation

   MacOS: Rightclick on the KNIME Analytics Platform installation and `Show Package Contents`, open the folder `Eclipse`

2. `plugins → <plugin_name> → bin`
3. The log files are `create_env.err` and `create_env.out`

Windows-specific issues

- Installation fails - potential issue: the installation folder of the KNIME Analytics Platform has a long path. Windows' long path limitations can be circumvented by enabling long path support as outlined here: https://docs.microsoft.com/en-us/windows/win32/fileio/maximum-file-path-limitation?tabs=registry
- Installation fails and the `create_env.err` contains `Access is denied.` - potential issue: Python processes are in the background. Killing the processes can solve this issue.

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