
KNIME Python API

Carsten Haubold, Adrian Nembach, Marcel Wiedenmann, Benjamin

Jan 31, 2023

CONTENTS

1	Contents	3
---	----------	---

This document lists the Python API that can be used to communicate with KNIME within *Python scripts* and *Python extensions*.

CONTENTS

1.1 Python Script API

This section lists the API of the module `knime.scripting.io` that functions as the main contact point between KNIME and Python in the [KNIME Python Script node](#). Please refer to the [KNIME Python Integration Guide](#) for more details on how to set up and use the node.

Note: Before KNIME AP 4.7, the module used to interact with KNIME from Python was called `knime_io` and provided a slightly different API. Since KNIME AP 4.7 the new Python Script node is no longer in *Labs* status and uses the `knime.scripting.io` module for interaction between KNIME and Python. It uses the same Table and Batch classes as can be used in KNIME Python Extensions. The previous API is described in [Deprecated Python Script API](#)

1.1.1 Inputs and outputs

These properties can be used to retrieve data from or pass data back to KNIME Analytics Platform. The length of the input and output lists depends on the number of input and output ports of the node.

Example: If you have a Python Script node configured with two input tables and one input object, you can access the two tables via `knime.scripting.io.input_tables[0]` and `knime.scripting.io.input_tables[1]`, and the input object via `knime.scripting.io.input_objects[0]`.

Input and output variables used to communicate with KNIME from within KNIME's Python Scripting nodes

`knime.scripting.io.flow_variables:` `Dict[str, Any] = {}`

A dictionary of flow variables provided by the KNIME workflow. New flow variables can be added to the output of the node by adding them to the dictionary. Supported flow variable types are numbers, strings, booleans and lists thereof.

`knime.scripting.io.input_objects:` `List =`
<`knime.scripting._io_containers._FixedSizeListView` object>

A list of input objects of this script node using zero-based indices. This list has a fixed size, which is determined by the number of input object ports configured for this node. Input objects are Python objects that are passed in from another Python script node's ``output_object`` port. This can, for instance, be used to pass trained models between Python nodes. If no input is given, the list exists but is empty.

`knime.scripting.io.input_tables:` `List[Table] =`
<`knime.scripting._io_containers._FixedSizeListView` object>

The input tables of this script node. This list has a fixed size, which is determined by the number of input table ports configured for this node. Tables are available in the same order as the port connectors are displayed alongside the node (from top to bottom), using zero-based indexing. If no input is given, the list exists but is empty.

knime.scripting.io.output_images: List =
<knime.scripting._io_containers._FixedSizeListView object>

The output images of this script node. This list has a fixed size, which is determined by the number of output images configured for this node. The value passed to the output port should be a bytes-like object encoding an SVG or PNG image.

Example:

```
import knime.scripting.io as knio

data = knio.input_tables[0].to_pandas()
buffer = io.BytesIO()

pyplot.figure()
pyplot.plot('x', 'y', data=data)
pyplot.savefig(buffer, format='svg')

knio.output_images[0] = buffer.getvalue()
```

knime.scripting.io.output_objects: List =
<knime.scripting._io_containers._FixedSizeListView object>

The output objects of this script node. This list has a fixed size, which is determined by the number of output object ports configured for this node. Each output object can be an arbitrary Python object as long as it can be *pickled*. Use this to, for example, pass a trained model to another Python script node.

Example:

```
model = torchvision.models.resnet18()
...
# train/finetune model
...
knime.scripting.io.output_objects[0] = model
```

knime.scripting.io.output_tables: List[Table | BatchOutputTable] =
<knime.scripting._io_containers._FixedSizeListView object>

The output tables of this script node. This list has a fixed size, which is determined by the number of output table ports configured for this node. You should assign a Table or BatchOutputTable to each output port of this node.

Example:

```
import knime.scripting.io as knio
knio.output_tables[0] = knio.Table.from_pandas(my_pandas_df)
```

knime.scripting.io.output_view: NodeView | None = None

The output view of the script node. This variable must be populated with a NodeView when using the Python View node. Views can be created by calling the `view(obj)` method with a viewable object. See the documentation of `view(obj)` to understand how views are created from different kinds of objects.

Example:

```
import knime.scripting.io as knio
import plotly.express as px
```

(continues on next page)

(continued from previous page)

```
fig = px.scatter(x=data_x, y=data_y)
knio.output_view = knio.view(fig)
```

1.1.2 Classes

class knime.scripting.io.Table

This class serves as public API to create KNIME tables either from pandas or pyarrow. These tables can then be sent back to KNIME. This class has to be instantiated by calling either `from_pyarrow()` or `from_pandas()`

__getitem__(*slicing: slice | List[int] | List[str] | Tuple[slice | List[int] | List[str], slice]*) → `_TabularView`

Creates a view of this Table by slicing rows and columns. The slicing syntax is similar to that of numpy arrays, but columns can also be addressed as index lists or via a list of column names.

The syntax is `[column_slice, row_slice]`. Note that this is the exact opposite order than in the deprecated scripting API's `ReadTable`.

Parameters

- **column_slice** – A column index, a column name, a slice object, a list of column indices, or a list of column names.
- **row_slice** – Optional: A slice object describing which rows to use.

Returns

A `_TabularView` representing a slice of the original Table

Example:

```
row_sliced_table = table[:, :100] # Get the first 100 rows
column_sliced_table = table[["name", "age"]] # Get all rows of the columns "name
↪ " and "age"
row_and_column_sliced_table = table[1:5, :100] # Get the first 100 rows of
↪ columns 1,2,3,4
```

batches() → `Iterator[Table]`

Returns a generator over the batches in this table. A batch is part of the table with all columns, but only a subset of the rows. A batch should always fit into memory (max size currently 64mb). The table being passed to `execute()` is already present in batches, so accessing the data this way is very efficient.

Example:

```
output_table = BatchOutputTable.create()
for batch in my_table.batches():
    input_batch = batch.to_pandas()
    # process the batch
    output_table.append(Table.from_pandas(input_batch))
```

static from_pandas(*data: pandas.DataFrame, sentinel: str | int | None = None, row_ids: str = 'auto'*)

Factory method to create a Table given a pandas.DataFrame. The index of the data frame will be used as RowKey by KNIME.

Example:

```
Table.from_pandas(my_pandas_df, sentinel="min")
```

Parameters

- **data** – A pandas.DataFrame
- **sentinel** – Interpret the following values in integral columns as missing value:
 - "min" min int32 or min int64 depending on the type of the column
 - "max" max int32 or max int64 depending on the type of the column
 - a special integer value that should be interpreted as missing value
- **row_ids** – Defines what RowID should be used. Must be one of the following values:
 - "keep": Keep the DataFrame.index as the RowID. Convert the index to strings if necessary.
 - "generate": Generate new RowIDs of the format f"Row{i}" where i is the position of the row (from 0 to length-1).
 - "auto": If the DataFrame.index is of type int or unsigned int, use f"Row{n}" where n is the index of the row. Else, use "keep".

static from_pyarrow(data: pyarrow.Table, sentinel: str | int | None = None, row_ids: str = 'auto')

Factory method to create a Table given a pyarrow.Table.

Example:

```
Table.from_pyarrow(my_pyarrow_table, sentinel="min")
```

Parameters

- **data** – A pyarrow.Table
- **sentinel** – Interpret the following values in integral columns as missing value:
 - "min" min int32 or min int64 depending on the type of the column
 - "max" max int32 or max int64 depending on the type of the column
 - a special integer value that should be interpreted as missing value
- **row_ids** – Defines what RowID should be used. Must be one of the following values:
 - "keep": Use the first column of the table as RowID. The first column must be of type string.
 - "generate": Generate new RowIDs of the format f"Row{i}" where i is the position of the row (from 0 to length-1).
 - "auto": Use the first column of the table if it has the name "<RowID>" and is of type string or integer.
 - * If the "<RowID>" column is of type string, use it directly
 - * If the "<RowID>" column is of an integer type use f"Row{n}" where n is the value of the integer column.
 - * Generate new RowIDs ("generate") if the first column has another type or name.

remove(slicing: str | int | List[str])

Implements remove method for Columnar data structures. The input can be a column index, a column name or a list of column names.

If the input is a column index, the column with that index will be removed. If it is a column name, then the first column with matching name is removed. Passing a list of column names will filter out all (including duplicate) columns with matching names.

Parameters

slicing – Can be of type integer representing the index in `column_names` to remove. Or a list of strings removing every column matching from that list. Or a string of which first occurrence is removed from the `column_names`.

Returns

A View missing the columns to be removed.

Raises

- **ValueError** if no matching column is found given a list or str –
- **IndexError** if column is accessed by integer and is out of bounds –
- **TypeError** if the key is neither a integer nor a string or list of strings. –

abstract property schema: `Schema`

The schema of this table, containing column names, types, and potentially metadata

to_batches() → `Iterator[Table]`

Alias for `Table.batches()`

to_pandas(*sentinel: str | int | None = None*) → `pandas.DataFrame`

Access this table as a `pandas.DataFrame`.

Parameters

sentinel – Replace missing values in integral columns by the given value, one of:

- "min" min int32 or min int64 depending on the type of the column
- "max" max int32 or max int64 depending on the type of the column
- An integer value that should be inserted for each missing value

to_pyarrow(*sentinel: str | int | None = None*) → `pyarrow.Table`

Access this table as a `pyarrow.Table`.

Parameters

sentinel – Replace missing values in integral columns by the given value, one of:

- "min" min int32 or min int64 depending on the type of the column
- "max" max int32 or max int64 depending on the type of the column
- An integer value that should be inserted for each missing value

class `knime.scripting.io.BatchOutputTable`

An output table generated by combining smaller tables (also called batches).

All batches must have the same number, names and types of columns.

Does not provide means to continue to work with the data but is meant to be used as a return value of a Node's `execute()` method.

abstract append(*batch: Table | pandas.DataFrame | pyarrow.Table | pyarrow.RecordBatch*) → None

Append a batch to this output table. The first batch defines the structure of the table, and all subsequent batches must have the same number of columns, column names and column types.

Note: Keep in mind that the RowID will be handled according to the “row_ids” mode chosen in `BatchOutputTable.create`.

static create(*row_ids: str = 'keep'*)

Create an empty `BatchOutputTable`

Parameters

row_ids – Defines what RowID should be used. Must be one of the following values:

- “keep”:
 - For appending DataFrames: Keep the `DataFrame.index` as the RowID. Convert the index to strings if necessary.
 - For appending Arrow tables or record batches: Use the first column of the table as RowID. The first column must be of type string.
- “generate”: Generate new RowIDs of the format `f"Row{i}"`

static from_batches(*generator, row_ids: str = 'generate'*)

Create output table where each batch is provided by a generator

Parameters

row_ids – See `BatchOutputTable.create`.

abstract property num_batches: int

The number of batches written to this output table

1.1.3 Views

`knime.scripting.io.view(obj)` → `NodeView`

Create an `NodeView` for the given object.

This method tries to find out the best option to display the given object. First, the method checks if a special view implementation (listed below) exists for the given object. Next, `IPython_repr_html_`, `_repr_svg_`, `_repr_png_`, or `_repr_jpeg_` are used.

Special view implementations:

- HTML: The obj must be of type str and start with “<!DOCTYPE html>”. The document must be self-contained and must not reference external resources. Links to external resources will be opened in an external browser.
- SVG: The obj must be of type str and contain a valid SVG
- PNG: The obj must be of type bytes and contain a PNG image file
- JPEG: The obj must be of type bytes and contain a JPEG image file
- Matplotlib: The obj must be a `matplotlib.figure.Figure`
- Plotly: The obj must be a `plotly.graph_objects.Figure`

Parameters

obj – The object which should be displayed

Raises

ValueError – If no view could be created for the given object

`knime.scripting.io.view_matplotlib(fig=None, format='png') → NodeView`

Create a view showing the given matplotlib figure.

The figure is displayed by exporting it as an SVG. If no figure is given the current active figure is displayed. Note that the figure is closed and should not be used after calling this method.

Parameters

- **fig** – A `matplotlib.figure.Figure` which should be displayed.
- **format** – Format of the view inside the HTML document. Either “png” or “svg”.

Raises

- **ImportError** – If matplotlib is not available.
- **TypeError** – If the figure is not a matplotlib figure.

`knime.scripting.io.view_seaborn() → NodeView`

Create a view showing the current active seaborn figure.

This function just calls `view_matplotlib()` because seaborn plots are just matplotlib figures under the hood.

Raises

ImportError – If matplotlib is not available.

`knime.scripting.io.view_plotly(fig) → NodeView`

Create a view showing the given plotly figure.

The figure is displayed by exporting it as an HTML document.

To be able to synchronize the selection between the view and other KNIME views the customdata 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])
node_view = view_plotly(fig)
```

Parameters

fig – A `plotly.graph_objects.Figure` object which should be displayed.

Raises

- **ImportError** – If plotly is not available.
- **TypeError** – If the figure is not a plotly figure.

`knime.scripting.io.view_html(html: str, svg_or_png: str | bytes | None = None, render_fn: Callable[[str | bytes] | None = None) → NodeView`

Create a NodeView that displays the given HTML document.

The document must be self-contained and must not reference external resources. Links to external resources will be opened in an external browser.

Parameters

- **html** – A string containing the HTML document.

- **svg_or_png** – A rendered representation of the HTML page. Either a string containing an SVG or a bytes object containing an PNG image
- **render_fn** – A callable that returns an SVG or PNG representation of the page

`knime.scripting.io.view_svg(svg: str) → NodeView`

Create a NodeView that displays the given SVG.

Parameters

svg – A string containing the SVG.

`knime.scripting.io.view_png(png: bytes) → NodeView`

Create a NodeView that displays the given PNG image.

Parameters

png – The bytes of the PNG image

`knime.scripting.io.view_jpeg(jpeg: bytes) → NodeView`

Create a NodeView that displays the given JPEG image.

Parameters

jpeg – The bytes of the JPEG image

`knime.scripting.io.view_ipy_repr(obj) → NodeView`

Create a NodeView by using the IPython `_repr_*` function of the object.

Tries to use `*_repr_html_*` `*_repr_svg_*` `*_repr_png_*` `*_repr_jpeg_*` in this order.

Parameters

obj – The object which should be displayed

Raises

ValueError – If no view could be created for the given object

class `knime.scripting.io.NodeView(html: str, svg_or_png: str | bytes | None = None, render_fn: Callable[[], str | bytes] | None = None)`

A view of a KNIME node that can be displayed for the user.

Do not create a NodeView directly but use the utility functions `view`, `view_html`, `view_svg`, `view_png`, and `view_jpeg`.

1.2 Python Extension Development (Labs)

These classes can be used by developers to implement their own Python nodes for KNIME. For a more detailed description see the [Pure Python Node Extensions Guide](#)

Note: Before KNIME AP 4.7, the module used to access KNIME functionality was called `knime_extension`. This module has been renamed to `knime.extension`.

1.2.1 Nodes

`class knime.extension.PythonNode`

Extend this class to provide a pure Python based node extension to KNIME Analytics Platform.

Users can either use the decorators `@knext.input_table`, `@knext.input_binary`, `@knext.output_table`, `@knext.output_binary`, and `@knext.output_view`, or populate the `input_ports`, `output_ports`, and `output_view` attributes.

Use the Python logging facilities and its `.warning` and `.error` methods to write warnings and errors to the KNIME console. `.info` and `.debug` will only show up in the KNIME console if the log level in KNIME is configured to show these.

Example:

```
import logging
import knime.extension as knext

LOGGER = logging.getLogger(__name__)

category = knext.category("/community", "mycategory", "My Category", "My category.
↳described", icon="icons/category.png")

@knext.node(name="Pure Python Node", node_type=knext.NodeType.LEARNER, icon_path=
↳"icons/icon.png", category=category)
@knext.input_table(name="Input Data", description="We read data from here")
@knext.output_table(name="Output Data", description="Whatever the node has produced
↳")
class TemplateNode(knext.PythonNode):
    # A Python node has a description.

    def configure(self, configure_context, table_schema):
        LOGGER.info(f"Configuring node")
        return table_schema

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

abstract `configure`(*config_context: ConfigurationContext, *inputs*)

Configure this Python node.

Parameters

- **config_context** – The ConfigurationContext providing KNIME utilities during execution
- ***inputs** – Each input table spec or binary port spec will be added as parameter, in the same order that the ports were defined.

Returns

Either a single spec, or a tuple or list of specs. The number of specs must match the number of defined output ports, and they must be returned in this order. Alternatively, instead of a spec, a `knext.Column` can be returned (if the spec shall only consist of one column).

Raises

InvalidParametersError – If the current input parameters do not satisfy this node's requirements.

abstract `execute(exec_context: ExecutionContext, *inputs)`

Execute this Python node.

Parameters

- **exec_context** – The ExecutionContext providing KNIME utilities during execution
- ***inputs** – Each input table or binary port object will be added as parameter, in the same order that the ports were defined. Tables will be provided as a *kn.Table*, while binary data will be a plain Python *bytes* object.

Returns

Either a single output object (table or binary), or a tuple or list of objects. The number of output objects must match the number of defined output ports, and they must be returned in this order. Tables must be provided as a *kn.Table* or *kn.BatchOutputTable*, while binary data should be returned as plain Python *bytes* object.

A node is part of a category:

`knime.extension.category(path: str, level_id: str, name: str, description: str, icon: str, after: str = "", locked: bool = True)`

Register a new node category.

A node category must only be created once. Use a string encoding the absolute category path to add nodes to an existing category.

Parameters

- **path** (*Union[str, Category]*) – The absolute “path” that lead to this category e.g. “/io/read”. The segments are the category level-IDs, separated by a slash (“/”). Categories that contain community nodes should be placed in the “/community” category.
- **level_id** (*str*) – The identifier of the level which is used as a path-segment and must be unique at the level specified by “path”.
- **name** (*str*) – The name of this category e.g. “File readers”.
- **description** (*str*) – A short description of the category.
- **icon** (*str*) – File path to 16x16 pixel PNG icon for this category. The path must be relative to the root of the extension.
- **after** (*str, optional*) – Specifies the level-id of the category after which this category should be sorted in. Defaults to “”.
- **locked** (*bool, optional*) – Set this to False to allow extensions from other vendors to add sub-categories or nodes to this category. Defaults to True.

Returns

The full path of the category which can be used to create nodes inside this category.

Return type

str

A node has a type:

class `knime.extension.NodeType(value)`

Defines the different node types that are available for Python based nodes.

LEARNER = 'Learner'

A node learning a model that is typically consumed by a PREDICTOR.

MANIPULATOR = 'Manipulator'

A node that manipulates data.

OTHER = 'Other'

A node that doesn't fit one of the other node types.

PREDICTOR = 'Predictor'

A node that predicts something typically using a model provided by a LEARNER.

SINK = 'Sink'

A node consuming data.

SOURCE = 'Source'

A node producing data.

VISUALIZER = 'Visualizer'

A node that visualizes data.

A node's configure method receives a configuration context that lets you interact with KNIME

class knime.extension.ConfigurationContext(*java_config_ctx, flow_variables*)

The ConfigurationContext provides utilities to communicate with KNIME during a node's configure() method.

property flow_variables: Dict[str, Any]

The flow variables coming in from KNIME as a dictionary with string keys. The dictionary can be edited and supports flow variables of the following types:

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

set_warning(*message: str*) → None

Sets a warning on the node.

Parameters

message – the warning message to display on the node

A node's execute method receives an execution context that lets you interact with KNIME and e.g. check whether the user has cancelled the execution of your Python node.

class knime.extension.ExecutionContext(*java_ctx, flow_variables*)

The ExecutionContext provides utilities to communicate with KNIME during a node's execute() method.

property flow_variables: Dict[str, Any]

The flow variables coming in from KNIME as a dictionary with string keys. The dictionary can be edited and supports flow variables of the following types:

- bool
- list(bool)

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

is_canceled() → bool

Returns true if this node's execution has been canceled from KNIME. Nodes can check for this property and return early if the execution does not need to finish. Raising a `RuntimeError` in that case is encouraged.

set_progress(*progress: float, message: str | None = None*)

Set the progress of the execution.

Note that the progress that can be set here is 80% of the total progress of a node execution. The first and last 10% are reserved for data transfer and will be set by the framework.

Parameters

- **progress** – a floating point number between 0.0 and 1.0
- **message** – an optional message to display in KNIME with the progress

set_warning(*message: str*) → None

Sets a warning on the node.

Parameters

message – the warning message to display on the node

Decorators

These decorators can be used to easily configure your Python node.

knime.extension.node(*name: str, node_type: NodeType, icon_path: str, category: str, after: str | None = None, id: str | None = None, is_deprecated: bool = False*) → Callable

Use this decorator to annotate a `PythonNode` class or function that creates a `PythonNode` instance that should correspond to a node in KNIME.

knime.extension.input_binary(*name: str, description: str, id: str*)

Use this decorator to define a bytes-serialized port object input of a node.

Parameters

- **name** – The name of the input port
- **description** – A description of the input port.
- **id** – A unique ID identifying the type of the Port. Only Ports with equal ID can be connected in KNIME

knime.extension.input_table(*name: str, description: str*)

Use this decorator to define an input port of type “Table” of a node.

knime.extension.output_binary(*name: str, description: str, id: str*)

Use this decorator to define a bytes-serialized port object output of a node.

Parameters

- **name** –
- **description** –
- **id** – A unique ID identifying the type of the Port. Only Ports with equal ID can be connected in KNIME

`knime.extension.output_table(name: str, description: str)`

Use this decorator to define an output port of type “Table” of a node.

`knime.extension.output_view(name: str, description: str, static_resources: str | None = None)`

Use this decorator to specify that this node produces a view

Parameters

- **name** – The name of the view
- **description** – Description of the view
- **static_resources** – The path to a folder of resources that will be available to the HTML page. The path given here must be relative to the root of the extension. The resources can be accessed by the same relative file path (e.g. “{static_resources}/{filename}”).

Parameters

To add parameterization to your nodes, the configuration dialog can be defined and customized. Each parameter can be used in the nodes execution by accessing `self.param_name`. These parameters can be set up by using the following parameter types. For a more detailed description see [Defining the node’s configuration dialog](#).

```
class knime.extension.IntParameter(label: str | None = None, description: str | None = None, default_value:
    int | Callable[[Version], int] = 0, validator: Callable[[int], None] |
    None = None, min_value: int | None = None, max_value: int | None =
    None, since_version: Version | str | None = None)
```

Parameter class for primitive integer types.

```
class knime.extension.DoubleParameter(label: str | None = None, description: str | None = None,
    default_value: float | Callable[[Version], float] = 0.0, validator:
    Callable[[float], None] | None = None, min_value: float | None =
    None, max_value: float | None = None, since_version: Version | str |
    None = None)
```

Parameter class for primitive float types.

```
class knime.extension.BoolParameter(label: str | None = None, description: str | None = None,
    default_value: bool | Callable[[Version], bool] = False, validator:
    Callable[[bool], None] | None = None, since_version: Version | str |
    None = None)
```

Parameter class for primitive boolean types.

```
class knime.extension.StringParameter(label: str | None = None, description: str | None = None,
    default_value: str | Callable[[Version], str] = "", enum: List[str] |
    None = None, validator: Callable[[str], None] | None = None,
    since_version: Version | str | None = None)
```

Parameter class for primitive string types.

```
class knime.extension.ColumnParameter(label: str | None = None, description: str | None = None,
    port_index: int = 0, column_filter: Callable[[Column], bool] |
    None = None, include_row_key: bool = False,
    include_none_column: bool = False, since_version: str | None =
    None)
```

Parameter class for single columns.

```
class knime.extension.MultiColumnParameter(label: str | None = None, description: str | None = None,
                                           port_index: int | None = 0, column_filter:
                                           Callable[[Column], bool] | None = None, since_version:
                                           Version | str | None = None)
```

Parameter class for multiple columns.

```
class knime.extension.EnumParameter(label: str | None = None, description: str | None = None,
                                     default_value: str | Callable[[Version], str] | None = None, enum:
                                     EnumParameterOptions | None = None, validator: Callable[[str],
                                     None] | None = None, since_version: Version | str | None = None)
```

Parameter class for multiple-choice parameter types. Replicates and extends the enum functionality previously implemented as part of StringParameter.

A subclass of EnumParameterOptions should be provided as the enum parameter, which should contain class attributes of the form OPTION_NAME = (OPTION_LABEL, OPTION_DESCRIPTION). The corresponding option attributes can be accessed via MyOptions.OPTION_NAME.name, .label, and .description respectively.

The .name attribute of each option is used as the selection constant, e.g. MyOptions.OPTION_NAME.name == "OPTION_NAME".

Example:

```
class CoffeeOptions(EnumParameterOptions):
    CLASSIC = ("Classic", "The classic chocolatey taste, with notes of bitterness_
    ↪and wood.")
    FRUITY = ("Fruity", "A fruity taste, with notes of berries and citrus.")
    WATERY = ("Watery", "A watery taste, with notes of water and wetness.")

coffee_selection_param = knext.EnumParameter(
    label="Coffee Selection",
    description="Select the type of coffee you like to drink.",
    default_value=CoffeeOptions.CLASSIC.name,
    enum=CoffeeOptions,
)
```

```
class knime.extension.EnumParameterOptions(value)
```

A helper class for creating EnumParameter options, based on Python's Enum class.

Developers should subclass this class, and provide enumeration options as class attributes of the subclass, of the form OPTION_NAME = (OPTION_LABEL, OPTION_DESCRIPTION).

Enum option objects can be accessed as attributes of the EnumParameterOptions subclass, e.g. MyEnum.OPTION_NAME. Each option object has the following attributes:

- name: the name of the class attribute, e.g. "OPTION_NAME", which is used as the selection constant;
- label: the label of the option, displayed in the configuration dialogue of the node;
- description: the description of the option, used along with the label to generate a list of the available options in the Node Description and in the configuration dialogue of the node.

Example:

```
class CoffeeOptions(EnumParameterOptions):
    CLASSIC = ("Classic", "The classic chocolatey taste, with notes of bitterness_
```

(continues on next page)

(continued from previous page)

```
↪ and wood.")
FRUITY = ("Fruity", "A fruity taste, with notes of berries and citrus.")
WATERY = ("Watery", "A watery taste, with notes of water and wetness.")
```

classmethod get_all_options()

Returns a list of all options defined in the EnumParameterOptions subclass.

Validation

While each parameter type listed above has default type validation (eg checking if the IntParameter contains only Integers), they also support custom validation via a property-like decorator notation. For instance, this can be used to verify that the parameter value matches a certain criteria (see example below). The validator should be placed below the definition of the corresponding parameter.

```
class knime.extension.IntParameter(label: str | None = None, description: str | None = None, default_value:
    int | Callable[[Version], int] = 0, validator: Callable[[int], None] |
    None = None, min_value: int | None = None, max_value: int | None =
    None, since_version: Version | str | None = None)
```

Parameter class for primitive integer types.

validator(func)

To be used as a decorator for setting a validator function for a parameter. Note that ‘func’ will be encapsulated in ‘_validator’ and will not be available in the namespace of the class.

Example:

```
@knext.node(args)
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(args):
        pass

    def execute(args):
        pass
```

Parameter Groups

Additionally these parameters can be combined in **parameter_groups**. These groups are visualized as sections in the configuration dialog. 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.

```
knime.extension.parameter_group(label: str, since_version: Version | str | None = None)
```

Used for injecting descriptor protocol methods into a custom parameter group class. “obj” in this context is the parameterized object instance or a parameter group instance.

Group validators need to raise an exception if a values-based condition is violated, where values is a dictionary of parameter names and values. Group validators can be set using either of the following methods:

- By implementing the “validate(self, values)” method inside the class definition of the group.

Example:

```
def validate(self, values):
    assert values['first_param'] + values['second_param'] < 100
```

- By using the “@group_name.validator” decorator notation inside the class definition of the “parent” of the group. The decorator has an optional ‘override’ parameter, set to True by default, which overrides the “validate” method. If ‘override’ is set to False, the “validate” method, if defined, will be called first.

Example:

```
@hyperparameters.validator(override=False)
def validate_hyperparams(values):
    assert values['first_param'] + values['second_param'] < 100
```

Example:

```
@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(args)
class MyNodeWithSettings:
    settings = MySettings()
    def configure(args):
        pass

    def execute(args):
        pass
```

1.2.2 Tables

Table and Schema are the two classes that are used to communicate tabular data (Table) during execute, or the table structure (Schema) in configure between Python and KNIME.

class knime.extension.Table

This class serves as public API to create KNIME tables either from pandas or pyarrow. These tables can then be sent back to KNIME. This class has to be instantiated by calling either `from_pyarrow()` or `from_pandas()`

__getitem__(*slicing: slice | List[int] | List[str] | Tuple[slice | List[int] | List[str], slice]*) → `_TabularView`

Creates a view of this Table by slicing rows and columns. The slicing syntax is similar to that of numpy arrays, but columns can also be addressed as index lists or via a list of column names.

The syntax is `[column_slice, row_slice]`. Note that this is the exact opposite order than in the deprecated scripting API's `ReadTable`.

Parameters

- **column_slice** – A column index, a column name, a slice object, a list of column indices, or a list of column names.
- **row_slice** – Optional: A slice object describing which rows to use.

Returns

A `_TabularView` representing a slice of the original Table

Example:

```
row_sliced_table = table[:, :100] # Get the first 100 rows
column_sliced_table = table[["name", "age"]] # Get all rows of the columns
→ "name" and "age"
row_and_column_sliced_table = table[1:5, :100] # Get the first 100 rows of
→ columns 1,2,3,4
```

batches() → `Iterator[Table]`

Returns a generator over the batches in this table. A batch is part of the table with all columns, but only a subset of the rows. A batch should always fit into memory (max size currently 64mb). The table being passed to `execute()` is already present in batches, so accessing the data this way is very efficient.

Example:

```
output_table = BatchOutputTable.create()
for batch in my_table.batches():
    input_batch = batch.to_pandas()
    # process the batch
    output_table.append(Table.from_pandas(input_batch))
```

static from_pandas(*data: pandas.DataFrame, sentinel: str | int | None = None, row_ids: str = 'auto'*)

Factory method to create a Table given a `pandas.DataFrame`. The index of the data frame will be used as `RowKey` by KNIME.

Example:

```
Table.from_pandas(my_pandas_df, sentinel="min")
```

Parameters

- **data** – A `pandas.DataFrame`
- **sentinel** – Interpret the following values in integral columns as missing value:
 - "min" min int32 or min int64 depending on the type of the column
 - "max" max int32 or max int64 depending on the type of the column
 - a special integer value that should be interpreted as missing value
- **row_ids** – Defines what `RowID` should be used. Must be one of the following values:
 - "keep": Keep the `DataFrame.index` as the `RowID`. Convert the index to strings if necessary.
 - "generate": Generate new `RowIDs` of the format `f"Row{i}"` where `i` is the position of the row (from 0 to `length-1`).

- "auto": If the `DataFrame.index` is of type `int` or `unsigned int`, use `f"Row{n}"` where `n` is the index of the row. Else, use "keep".

static from_pyarrow(*data: pyarrow.Table, sentinel: str | int | None = None, row_ids: str = 'auto'*)

Factory method to create a `Table` given a `pyarrow.Table`.

Example:

```
Table.from_pyarrow(my_pyarrow_table, sentinel="min")
```

Parameters

- **data** – A `pyarrow.Table`
- **sentinel** – Interpret the following values in integral columns as missing value:
 - "min" min `int32` or min `int64` depending on the type of the column
 - "max" max `int32` or max `int64` depending on the type of the column
 - a special integer value that should be interpreted as missing value
- **row_ids** – Defines what RowID should be used. Must be one of the following values:
 - "keep": Use the first column of the table as RowID. The first column must be of type string.
 - "generate": Generate new RowIDs of the format `f"Row{i}"` where `i` is the position of the row (from 0 to `length-1`).
 - "auto": Use the first column of the table if it has the name "<RowID>" and is of type string or integer.
 - * If the "<RowID>" column is of type string, use it directly
 - * If the "<RowID>" column is of an integer type use `f"Row{n}"` where `n` is the value of the integer column.
 - * Generate new RowIDs ("generate") if the first column has another type or name.

remove(*slicing: str | int | List[str]*)

Implements remove method for Columnar data structures. The input can be a column index, a column name or a list of column names.

If the input is a column index, the column with that index will be removed. If it is a column name, then the first column with matching name is removed. Passing a list of column names will filter out all (including duplicate) columns with matching names.

Parameters

slicing – Can be of type integer representing the index in `column_names` to remove. Or a list of strings removing every column matching from that list. Or a string of which first occurrence is removed from the `column_names`.

Returns

A View missing the columns to be removed.

Raises

- **ValueError** if no matching column is found given a list or str –
- **IndexError** if column is accessed by integer and is out of bounds –

- **TypeError** if the key is neither a integer nor a string or list of strings. –

abstract property schema: `Schema`

The schema of this table, containing column names, types, and potentially metadata

to_batches() → `Iterator[Table]`

Alias for `Table.batches()`

to_pandas(*sentinel: str | int | None = None*) → `pandas.DataFrame`

Access this table as a `pandas.DataFrame`.

Parameters

sentinel – Replace missing values in integral columns by the given value, one of:

- "min" min int32 or min int64 depending on the type of the column
- "max" max int32 or max int64 depending on the type of the column
- An integer value that should be inserted for each missing value

to_pyarrow(*sentinel: str | int | None = None*) → `pyarrow.Table`

Access this table as a `pyarrow.Table`.

Parameters

sentinel – Replace missing values in integral columns by the given value, one of:

- "min" min int32 or min int64 depending on the type of the column
- "max" max int32 or max int64 depending on the type of the column
- An integer value that should be inserted for each missing value

class `knime.extension.BatchOutputTable`

An output table generated by combining smaller tables (also called batches).

All batches must have the same number, names and types of columns.

Does not provide means to continue to work with the data but is meant to be used as a return value of a Node's `execute()` method.

abstract append(*batch: Table | pandas.DataFrame | pyarrow.Table | pyarrow.RecordBatch*) → `None`

Append a batch to this output table. The first batch defines the structure of the table, and all subsequent batches must have the same number of columns, column names and column types.

Note: Keep in mind that the RowID will be handled according to the "row_ids" mode chosen in `BatchOutputTable.create`.

static create(*row_ids: str = 'keep'*)

Create an empty `BatchOutputTable`

Parameters

row_ids – Defines what RowID should be used. Must be one of the following values:

- "keep":
 - For appending DataFrames: Keep the `DataFrame.index` as the RowID. Convert the index to strings if necessary.
 - For appending Arrow tables or record batches: Use the first column of the table as RowID. The first column must be of type string.

- "generate": Generate new RowIDs of the format f"Row{i}"

static from_batches(generator, row_ids: str = 'generate')

Create output table where each batch is provided by a generator

Parameters

row_ids – See BatchOutputTable.create.

abstract property num_batches: int

The number of batches written to this output table

class knime.extension.**Schema**(ktypes: List[KnimeType | Type], names: List[str], metadata: List | None = None)

A schema defines the data types and names of the columns inside a table. Additionally, it can hold metadata for the individual columns.

__getitem__(slicing: slice | List[int] | List[str]) → _ColumnarView

Creates a view of this Table or Schema by slicing columns. The slicing syntax is similar to that of numpy arrays, but columns can also be addressed as index lists or via a list of column names.

Parameters

column_slice – A column index, a column name, a slice object, a list of column indices, or a list of column names. For single indices, the view will create a “Column” object. For slices or lists of indices, a new Schema will be returned.

Returns

A _ColumnarView representing a slice of the original Schema or Table.

Examples:

Get columns 1,2,3,4: sliced_schema = schema[1:5]

Get the columns “name” and “age”: sliced_schema = schema[["name", "age"]]

property column_names: List[str]

Return the list of column names

classmethod deserialize(table_schema: dict) → Schema

Construct a Schema from a dict that was retrieved from KNIME in JSON encoded form as the input to a node’s configure() method.

KNIME provides table information with a RowKey column at the beginning, which we drop before returning the created schema.

classmethod from_columns(columns: Sequence[Column] | Column)

Create a schema from a single column or a list of columns

classmethod from_types(ktypes: List[KnimeType | Type], names: List[str], metadata: List | None = None)

Create a schema from a list of column data types, names and metadata

property num_columns

The number of columns in this schema

remove(slicing: str | int | List[str])

Implements remove method for Columnar data structures. The input can be a column index, a column name or a list of column names.

If the input is a column index, the column with that index will be removed. If it is a column name, then the first column with matching name is removed. Passing a list of column names will filter out all (including duplicate) columns with matching names.

Parameters

slicing – Can be of type integer representing the index in `column_names` to remove. Or a list of strings removing every column matching from that list. Or a string of which first occurrence is removed from the `column_names`.

Returns

A View missing the columns to be removed.

Raises

- **ValueError** if no matching column is found given a list or str –
- **IndexError** if column is accessed by integer and is out of bounds –
- **TypeError** if the key is neither a integer nor a string or list of strings. –

serialize() → Dict

Convert this Schema into dict which can then be JSON encoded and sent to KNIME as result of a node's `configure()` method.

Because KNIME expects a row key column as first column of the schema, but we don't include this in the KNIME Python table schema, we insert a row key column here.

Raises

RuntimeError – if duplicate column names are detected

class `knime.extension.Column(ktype: KnimeType | Type, name: str, metadata=None)`

A column inside a table schema consists of the knime datatype, a column name and optional metadata.

__init__(*ktype: KnimeType | Type, name: str, metadata=None*)

Construct a Column from type, name and optional metadata.

Parameters

- **ktype** – The KNIME type of the column or a type which can be converted via `knime.api.schema.logical(ktype)` to a KNIME type
- **name** – The name of the column. May not be empty.

Raises

- **TypeError** – if the type is no KNIME type or cannot be converted to a KNIME type
- **ValueError** – if the name is empty

Data Types

These are helper functions to create KNIME compatible datatypes. For instance, if a new column is created.

`knime.extension.int32()`

Create a KNIME integer type with 32 bits

`knime.extension.int64()`

Create a KNIME integer type with 64 bits

`knime.extension.double()`

Create a KNIME floating point type with double precision (64 bits)

`knime.extension.bool_()`

Create a KNIME boolean type

`knime.extension.string(dict_encoding_key_type: DictEncodingKeyType | None = None)`

Create a KNIME string type.

Parameters

dict_encoding_key_type – The key type to use for dictionary encoding. If this is `None` (the default), no dictionary encoding will be used. Dictionary encoding helps to reduce storage space and read/write performance for columns with repeating values such as categorical data.

`knime.extension.blob(dict_encoding_key_type: DictEncodingKeyType | None = None)`

Create a KNIME blob type for binary data of variable length

Parameters

dict_encoding_key_type – The key type to use for dictionary encoding. If this is `None` (the default), no dictionary encoding will be used. Dictionary encoding helps to reduce storage space and read/write performance for columns with repeating values such as categorical data.

`knime.extension.list_(inner_type: KnimeType)`

Create a KNIME type that is a list of the given inner types

Parameters

inner_type – The type of the elements in the list. Must be a `KnimeType`

`knime.extension.struct(*inner_types)`

Create a KNIME structured data type where each given argument represents a field of the struct.

Parameters

inner_types – The argument list of this method defines the fields in this structured data type. Each inner type must be a KNIME type

`knime.extension.logical(value_type) → LogicalType`

Create a KNIME logical data type of the given Python value type.

Parameters

value_type – The type of the values inside this column. A `knime.api.types.PythonValueFactory` must be registered for this type.

Raises

TypeError – if no `PythonValueFactory` has been registered for this value type with `knime.api.types.register_python_value_factory`

1.3 Deprecated Python Script API

This section lists the API of the module `knime_io` that functioned as the main contact point between KNIME and Python in the [KNIME Python Script node](#) in KNIME AP before version 4.7, when the Python Script node was moved out of Labs. Please refer to the [KNIME Python Integration Guide](#) for more details on how to set up and use the node.

Warning: This API is deprecated since KNIME AP 4.7, please use the current API as described in [Python Script API](#)

1.3.1 Inputs and outputs

These properties can be used to retrieve data from or pass data back to KNIME Analytics Platform. The length of the input and output lists depends on the number of input and output ports of the node.

Example: If you have a Python Script node configured with two input tables and one input object, you can access the two tables via `knime_io.input_tables[0]` and `knime_io.input_tables[1]`, and the input object via `knime_io.input_objects[0]`.

knime_io.flow_variables: Dict[str, Any] = {}

A dictionary of flow variables provided by the KNIME workflow. New flow variables can be added to the output of the node by adding them to the dictionary. Supported flow variable types are numbers, strings, booleans and lists thereof.

knime_io.input_objects: List = <knime.scripting._io_containers._FixedSizeListView object>

A list of input objects of this script node using zero-based indices. This list has a fixed size, which is determined by the number of input object ports configured for this node. Input objects are Python objects that are passed in from another Python script node's ``output_object`` port. This can, for instance, be used to pass trained models between Python nodes. If no input is given, the list exists but is empty.

knime_io.input_tables: List[ReadTable] = <knime.scripting._io_containers._FixedSizeListView object>

The input tables of this script node. This list has a fixed size, which is determined by the number of input table ports configured for this node. Tables are available in the same order as the port connectors are displayed alongside the node (from top to bottom), using zero-based indexing. If no input is given, the list exists but is empty.

knime_io.output_images: List = <knime.scripting._io_containers._FixedSizeListView object>

The output images of this script node. This list has a fixed size, which is determined by the number of output images configured for this node. The value passed to the output port should be an array of bytes encoding an SVG or PNG image.

Example:

```
data = knime_io.input_tables[0].to_pandas()
buffer = io.BytesIO()

pyplot.figure()
pyplot.plot('x', 'y', data=data)
pyplot.savefig(buffer, format='svg')

knime_io.output_images[0] = buffer.getvalue()
```

knime_io.output_objects: List = <knime.scripting._io_containers._FixedSizeListView object>

The output objects of this script node. This list has a fixed size, which is determined by the number of output object ports configured for this node. Each output object can be an arbitrary Python object as long as it can be *pickled*. Use this to, for example, pass a trained model to another Python script node.

Example:

```
model = torchvision.models.resnet18()
...
```

(continues on next page)

(continued from previous page)

```
# train/finetune model
...
knime_io.output_objects[0] = model
```

`knime_io.output_tables: List[WriteTable] =`
`<knime.scripting._io_containers._FixedSizeListView object>`

The output tables of this script node. This list has a fixed size, which is determined by the number of output table ports configured for this node. You should assign a `WriteTable` or `BatchWriteTable` to each output port of this node. See the factory methods `knime_io.write_table()` and `knime_io.batch_write_table()` below.

Example:

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

1.3.2 Factory methods

Use these methods to fill the `knime_io.output_tables`.

`knime_io.batch_write_table()` → `BatchWriteTable`

Factory method to create an empty `BatchWriteTable` that can be filled sequentially batch by batch (see Example).

Example:

```
table = knime_io.batch_write_table()
table.append(df_1)
table.append(df_2)
knime_io.output_tables[0] = table
```

Warning: This class is deprecated since KNIME AP 4.7, use `knime.api.table.BatchOutputTable.create()` instead.

`knime_io.write_table(data: ReadTable | pandas.DataFrame | pyarrow.Table, sentinel: str | int | None = None)`
 → `WriteTable`

Factory method to create a `WriteTable` given a `pandas.DataFrame` or a `pyarrow.Table`. If the input is a `pyarrow.Table`, its first column must contain unique row identifiers of type 'string'.

Example:

```
knime_io.output_tables[0] = knime_io.write_table(my_pandas_df, sentinel="min")
```

Parameters

- **data** – A `ReadTable`, `pandas.DataFrame` or a `pyarrow.Table`
- **sentinel** – Interpret the following values in integral columns as missing value:
 - "min" min int32 or min int64 depending on the type of the column
 - "max" max int32 or max int64 depending on the type of the column
 - a special integer value that should be interpreted as missing value

Warning: This method is deprecated since KNIME AP 4.7, use `knime.api.table.Table.from_pandas()` or `knime.api.table.Table.from_pyarrow()` instead.

1.3.3 Classes

class `knime.scripting._deprecated._table.Batch`

A batch is a part of a table containing data. A batch should always fit into system memory, thus all methods accessing the data will be processed immediately and synchronously.

It can be sliced before the data is accessed as `pandas.DataFrame` or `pyarrow.RecordBatch`.

__getitem__ (*slicing: slice | Tuple[slice, slice | List[int] | List[str]]*) → `SlicedDataView`

Creates a view of this batch by slicing specific rows and columns. The slicing syntax is similar to that of numpy arrays, but columns can also be addressed as index lists or via a list of column names.

Parameters

- **row_slice** – A slice object describing which rows to use.
- **column_slice** – Optional. A slice object, a list of column indices, or a list of column names.

Returns

A `SlicedDataView` that can be converted to `pandas` or `pyarrow`.

Example:

```
full_batch = batch[:] # Slice/Get the full batch

# Slicing works for rows and columns. Column slices can be defined with int's,
↪ or the column names
row_sliced_batch = batch[:100] # Get first 100 rows of the batch
column_sliced_batch = batch[:, ["name", "age"]] # Get all rows of the columns
↪ "name" and "age"
row_and_column_sliced_batch = batch[:100, 1:5] # Get the first 100 rows of,
↪ columns 1,2,3,4

# The resulting `sliced_batches` cannot be sliced further. But they can be,
↪ converted to pandas or pyarrow.
```

abstract property `column_names: Tuple[str, ...]`

Returns the list of column names.

abstract property `num_columns: int`

Returns the number of columns in the table.

abstract property `num_rows: int`

Returns the number of rows in the table.

If the table is not completely available yet because batches are still appended to it, querying the number of rows blocks until all data is available.

property `shape: Tuple[int, int]`

Returns a tuple in the form (numRows, numColumns) representing the shape of this table.

If the table is not completely available yet because batches are still appended to it, querying the shape blocks until all data is available.

abstract to_pandas(*sentinel*: str | int | None = None) → pandas.DataFrame

Access the batch or table as a pandas.DataFrame.

Parameters

sentinel – Replace missing values in integral columns by the given value, one of:

- "min" min int32 or min int64 depending on the type of the column
- "max" max int32 or max int64 depending on the type of the column
- An integer value that should be inserted for each missing value

Raises

IndexError – If rows or columns were requested outside of the available shape

abstract to_pyarrow(*sentinel*: str | int | None = None) → pyarrow.RecordBatch | pyarrow.Table

Access this batch or table as a pyarrow.RecordBatch or pyarrow.table. The returned type depends on the type of the underlying object. When called on a ReadTable, returns a pyarrow.Table.

Parameters

sentinel – Replace missing values in integral columns by the given value, one of:

- "min" min int32 or min int64 depending on the type of the column
- "max" max int32 or max int64 depending on the type of the column
- An integer value that should be inserted for each missing value

Raises

IndexError – If rows or columns were requested outside of the available shape

class knime.scripting._deprecated._table.ReadTable

A KNIME ReadTable provides access to the data provided from KNIME, either in full (must fit into memory) or split into row-wise batches.

Warning: This class is deprecated since KNIME AP 4.7, use `knime.api.table.Table` instead.

__getitem__(*slicing*: slice | Tuple[slice, slice | List[int] | List[str]]) → SlicedDataView

Creates a view of this ReadTable by slicing rows and columns. The slicing syntax is similar to that of numpy arrays, but columns can also be addressed as index lists or via a list of column names.

The returned *sliced_table* cannot be sliced further. But they can be converted to pandas or pyarrow.

Parameters

- **row_slice** – A slice object describing which rows to use.
- **column_slice** – Optional. A slice object, a list of column indices, or a list of column names.

Returns

a SlicedDataView that can be converted to pandas or pyarrow.

Example:

```
row_sliced_table = table[:100] # Get the first 100 rows
column_sliced_table = table[:, ["name", "age"]] # Get all rows of the columns
→ "name" and "age"
row_and_column_sliced_table = table[:100, 1:5] # Get the first 100 rows of
→ columns 1,2,3,4
```

(continues on next page)

(continued from previous page)

```
df = row_and_column_sliced_table.to_pandas()
```

__len__() → int

Returns the number of batches of this table

abstract batches() → Iterator[Batch]

Returns an generator for the batches in this table. If the generator is advanced to a batch that is not available yet, it will block until the data is present. `len(my_read_table)` gives the static amount of batches within the table, which is not updated.

Example:

```
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)
```

abstract property column_names: Tuple[str, ...]

Returns the list of column names.

abstract property num_batches: int

Returns the number of batches in this table.

If the table is not completely available yet because batches are still appended to it, querying the number of batches blocks until all data is available.

abstract property num_columns: int

Returns the number of columns in the table.

abstract property num_rows: int

Returns the number of rows in the table.

If the table is not completely available yet because batches are still appended to it, querying the number of rows blocks until all data is available.

property shape: Tuple[int, int]

Returns a tuple in the form (numRows, numColumns) representing the shape of this table.

If the table is not completely available yet because batches are still appended to it, querying the shape blocks until all data is available.

abstract to_pandas(*sentinel*: str | int | None = None) → pandas.DataFrame

Access the batch or table as a pandas.DataFrame.

Parameters

sentinel – Replace missing values in integral columns by the given value, one of:

- "min" min int32 or min int64 depending on the type of the column
- "max" max int32 or max int64 depending on the type of the column
- An integer value that should be inserted for each missing value

Raises

IndexError – If rows or columns were requested outside of the available shape

abstract to_pyarrow(*sentinel*: *str* | *int* | *None* = *None*) → *pyarrow.RecordBatch* | *pyarrow.Table*

Access this batch or table as a *pyarrow.RecordBatch* or *pyarrow.table*. The returned type depends on the type of the underlying object. When called on a *ReadTable*, returns a *pyarrow.Table*.

Parameters

sentinel – Replace missing values in integral columns by the given value, one of:

- "min" min int32 or min int64 depending on the type of the column
- "max" max int32 or max int64 depending on the type of the column
- An integer value that should be inserted for each missing value

Raises

IndexError – If rows or columns were requested outside of the available shape

class *knime.scripting._deprecated._table.WriteTable*

A table that can be filled as a whole.

Warning: This class is deprecated since KNIME AP 4.7, use *knime.api.table.Table* instead.

abstract property column_names: *Tuple*[*str*, ...]

Returns the list of column names.

abstract property num_batches: *int*

Returns the number of batches in this table.

If the table is not completely available yet because batches are still appended to it, querying the number of batches blocks until all data is available.

abstract property num_columns: *int*

Returns the number of columns in the table.

abstract property num_rows: *int*

Returns the number of rows in the table.

If the table is not completely available yet because batches are still appended to it, querying the number of rows blocks until all data is available.

property shape: *Tuple*[*int*, *int*]

Returns a tuple in the form (numRows, numColumns) representing the shape of this table.

If the table is not completely available yet because batches are still appended to it, querying the shape blocks until all data is available.

class *knime.scripting._deprecated._table.BatchWriteTable*

A table that can be filled batch by batch.

Warning: This class is deprecated since KNIME AP 4.7, use *knime.api.table.BatchOutputTable* instead.

abstract append(*data*: *Batch* | *pandas.DataFrame* | *pyarrow.RecordBatch*, *sentinel*: *str* | *int* | *None* = *None*)

Appends a batch with the given data to the end of this table. The number of columns, as well as their data types, must match that of the previous batches in this table. Note that this cannot take a *pyarrow.Table* as input. With *pyarrow*, it can only process batches, which can be created as follows from some input table.

Example:

```

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)

```

Parameters

- **data** – A batch, a pandas.DataFrame or a pyarrow.RecordBatch
- **sentinel** – Only if data is a pandas.DataFrame or pyarrow.RecordBatch. Interpret the following values in integral columns as missing value:
 - "min" min int32 or min int64 depending on the type of the column
 - "max" max int32 or max int64 depending on the type of the column
 - a special integer value that should be interpreted as missing value

Raises

ValueError – If the new batch does not have the same columns as previous batches in this Writetable.

abstract property column_names: `Tuple[str, ...]`

Returns the list of column names.

static create() → BatchWriteTable

Create an empty BatchWriteTable

abstract property num_batches: `int`

Returns the number of batches in this table.

If the table is not completely available yet because batches are still appended to it, querying the number of batches blocks until all data is available.

abstract property num_columns: `int`

Returns the number of columns in the table.

abstract property num_rows: `int`

Returns the number of rows in the table.

If the table is not completely available yet because batches are still appended to it, querying the number of rows blocks until all data is available.

property shape: `Tuple[int, int]`

Returns a tuple in the form (numRows, numColumns) representing the shape of this table.

If the table is not completely available yet because batches are still appended to it, querying the shape blocks until all data is available.

Contents

Python Script API

This section lists the API of the module `knime.scripting.io` that functions as the main contact point between KNIME and Python in the [KNIME Python Script node](#). Please refer to the [KNIME Python Integration Guide](#) for more details on how to set up and use the node.

Note: Before KNIME AP 4.7, the module used to interact with KNIME from Python was called `knime_io` and provided a slightly different API. Since KNIME AP 4.7 the new Python Script node is no longer in *Labs* status and uses the `knime.scripting.io` module for interaction between KNIME and Python. It uses the same Table and Batch classes as can be used in KNIME Python Extensions. The previous API is described in [Deprecated Python Script API](#)

Inputs and outputs

These properties can be used to retrieve data from or pass data back to KNIME Analytics Platform. The length of the input and output lists depends on the number of input and output ports of the node.

Example: If you have a Python Script node configured with two input tables and one input object, you can access the two tables via `knime.scripting.io.input_tables[0]` and `knime.scripting.io.input_tables[1]`, and the input object via `knime.scripting.io.input_objects[0]`.

Input and output variables used to communicate with KNIME from within KNIME's Python Scripting nodes

`knime.scripting.io.flow_variables:` Dict[str, Any] = {}

A dictionary of flow variables provided by the KNIME workflow. New flow variables can be added to the output of the node by adding them to the dictionary. Supported flow variable types are numbers, strings, booleans and lists thereof.

`knime.scripting.io.input_objects:` List =
<`knime.scripting._io_containers._FixedSizeListView` object>

A list of input objects of this script node using zero-based indices. This list has a fixed size, which is determined by the number of input object ports configured for this node. Input objects are Python objects that are passed in from another Python script node's ``output_object`` port. This can, for instance, be used to pass trained models between Python nodes. If no input is given, the list exists but is empty.

`knime.scripting.io.input_tables:` List[Table] =
<`knime.scripting._io_containers._FixedSizeListView` object>

The input tables of this script node. This list has a fixed size, which is determined by the number of input table ports configured for this node. Tables are available in the same order as the port connectors are displayed alongside the node (from top to bottom), using zero-based indexing. If no input is given, the list exists but is empty.

`knime.scripting.io.output_images:` List =
<`knime.scripting._io_containers._FixedSizeListView` object>

The output images of this script node. This list has a fixed size, which is determined by the number of output images configured for this node. The value passed to the output port should be a bytes-like object encoding an SVG or PNG image.

Example:

```
import knime.scripting.io as knio

data = knio.input_tables[0].to_pandas()
buffer = io.BytesIO()

pyplot.figure()
pyplot.plot('x', 'y', data=data)
pyplot.savefig(buffer, format='svg')

knio.output_images[0] = buffer.getvalue()
```

knime.scripting.io.output_objects: List =
<knime.scripting._io_containers._FixedSizeListView object>

The output objects of this script node. This list has a fixed size, which is determined by the number of output object ports configured for this node. Each output object can be an arbitrary Python object as long as it can be *pickled*. Use this to, for example, pass a trained model to another Python script node.

Example:

```
model = torchvision.models.resnet18()
...
# train/finetune model
...
knime.scripting.io.output_objects[0] = model
```

knime.scripting.io.output_tables: List[Table | BatchOutputTable] =
<knime.scripting._io_containers._FixedSizeListView object>

The output tables of this script node. This list has a fixed size, which is determined by the number of output table ports configured for this node. You should assign a Table or BatchOutputTable to each output port of this node.

Example:

```
import knime.scripting.io as knio
knio.output_tables[0] = knio.Table.from_pandas(my_pandas_df)
```

knime.scripting.io.output_view: NodeView | None = None

The output view of the script node. This variable must be populated with a NodeView when using the Python View node. Views can be created by calling the `view(obj)` method with a viewable object. See the documentation of `view(obj)` to understand how views are created from different kinds of objects.

Example:

```
import knime.scripting.io as knio
import plotly.express as px

fig = px.scatter(x=data_x, y=data_y)
knio.output_view = knio.view(fig)
```

Classes

`class knime.scripting.io.Table`

This class serves as public API to create KNIME tables either from pandas or pyarrow. These tables can then be sent back to KNIME. This class has to be instantiated by calling either `from_pyarrow()` or `from_pandas()`

`__getitem__`(*slicing: slice | List[int] | List[str] | Tuple[slice | List[int] | List[str], slice]*) → `_TabularView`

Creates a view of this Table by slicing rows and columns. The slicing syntax is similar to that of numpy arrays, but columns can also be addressed as index lists or via a list of column names.

The syntax is `[column_slice, row_slice]`. Note that this is the exact opposite order than in the deprecated scripting API's `ReadTable`.

Parameters

- **`column_slice`** – A column index, a column name, a slice object, a list of column indices, or a list of column names.
- **`row_slice`** – Optional: A slice object describing which rows to use.

Returns

A `_TabularView` representing a slice of the original Table

Example:

```
row_sliced_table = table[:, :100] # Get the first 100 rows
column_sliced_table = table[["name", "age"]] # Get all rows of the columns
↳ "name" and "age"
row_and_column_sliced_table = table[1:5, :100] # Get the first 100 rows of
↳ columns 1,2,3,4
```

`batches()` → `Iterator[Table]`

Returns a generator over the batches in this table. A batch is part of the table with all columns, but only a subset of the rows. A batch should always fit into memory (max size currently 64mb). The table being passed to `execute()` is already present in batches, so accessing the data this way is very efficient.

Example:

```
output_table = BatchOutputTable.create()
for batch in my_table.batches():
    input_batch = batch.to_pandas()
    # process the batch
    output_table.append(Table.from_pandas(input_batch))
```

`static from_pandas`(*data: pandas.DataFrame, sentinel: str | int | None = None, row_ids: str = 'auto'*)

Factory method to create a Table given a pandas.DataFrame. The index of the data frame will be used as RowKey by KNIME.

Example:

```
Table.from_pandas(my_pandas_df, sentinel="min")
```

Parameters

- **`data`** – A pandas.DataFrame
- **`sentinel`** – Interpret the following values in integral columns as missing value:
 - "min" min int32 or min int64 depending on the type of the column

- "max" max int32 or max int64 depending on the type of the column
- a special integer value that should be interpreted as missing value
- **row_ids** – Defines what RowID should be used. Must be one of the following values:
 - "keep": Keep the `DataFrame.index` as the RowID. Convert the index to strings if necessary.
 - "generate": Generate new RowIDs of the format `f"Row{i}"` where `i` is the position of the row (from 0 to `length-1`).
 - "auto": If the `DataFrame.index` is of type `int` or `unsigned int`, use `f"Row{n}"` where `n` is the index of the row. Else, use "keep".

static from_pyarrow(*data: pyarrow.Table, sentinel: str | int | None = None, row_ids: str = 'auto'*)

Factory method to create a Table given a pyarrow.Table.

Example:

```
Table.from_pyarrow(my_pyarrow_table, sentinel="min")
```

Parameters

- **data** – A `pyarrow.Table`
- **sentinel** – Interpret the following values in integral columns as missing value:
 - "min" min int32 or min int64 depending on the type of the column
 - "max" max int32 or max int64 depending on the type of the column
 - a special integer value that should be interpreted as missing value
- **row_ids** – Defines what RowID should be used. Must be one of the following values:
 - "keep": Use the first column of the table as RowID. The first column must be of type string.
 - "generate": Generate new RowIDs of the format `f"Row{i}"` where `i` is the position of the row (from 0 to `length-1`).
 - "auto": Use the first column of the table if it has the name "<RowID>" and is of type string or integer.
 - * If the "<RowID>" column is of type string, use it directly
 - * If the "<RowID>" column is of an integer type use `f"Row{n}"` where `n` is the value of the integer column.
 - * Generate new RowIDs ("generate") if the first column has another type or name.

remove(*slicing: str | int | List[str]*)

Implements remove method for Columnar data structures. The input can be a column index, a column name or a list of column names.

If the input is a column index, the column with that index will be removed. If it is a column name, then the first column with matching name is removed. Passing a list of column names will filter out all (including duplicate) columns with matching names.

Parameters

- slicing** – Can be of type integer representing the index in `column_names` to remove. Or

a list of strings removing every column matching from that list. Or a string of which first occurrence is removed from the `column_names`.

Returns

A View missing the columns to be removed.

Raises

- **ValueError** if no matching column is found given a list or str –
- **IndexError** if column is accessed by integer and is out of bounds –
- **TypeError** if the key is neither a integer nor a string or list of strings. –

abstract property schema: `Schema`

The schema of this table, containing column names, types, and potentially metadata

to_batches() \rightarrow `Iterator[Table]`

Alias for `Table.batches()`

to_pandas(*sentinel: str | int | None = None*) \rightarrow `pandas.DataFrame`

Access this table as a `pandas.DataFrame`.

Parameters

sentinel – Replace missing values in integral columns by the given value, one of:

- "min" min int32 or min int64 depending on the type of the column
- "max" max int32 or max int64 depending on the type of the column
- An integer value that should be inserted for each missing value

to_pyarrow(*sentinel: str | int | None = None*) \rightarrow `pyarrow.Table`

Access this table as a `pyarrow.Table`.

Parameters

sentinel – Replace missing values in integral columns by the given value, one of:

- "min" min int32 or min int64 depending on the type of the column
- "max" max int32 or max int64 depending on the type of the column
- An integer value that should be inserted for each missing value

class `knime.scripting.io.BatchOutputTable`

An output table generated by combining smaller tables (also called batches).

All batches must have the same number, names and types of columns.

Does not provide means to continue to work with the data but is meant to be used as a return value of a Node's `execute()` method.

abstract append(*batch: Table | pandas.DataFrame | pyarrow.Table | pyarrow.RecordBatch*) \rightarrow `None`

Append a batch to this output table. The first batch defines the structure of the table, and all subsequent batches must have the same number of columns, column names and column types.

Note: Keep in mind that the RowID will be handled according to the "row_ids" mode chosen in `BatchOutputTable.create`.

static create(*row_ids: str = 'keep'*)

Create an empty BatchOutputTable

Parameters

row_ids – Defines what RowID should be used. Must be one of the following values:

- "keep":
 - For appending DataFrames: Keep the DataFrame.index as the RowID. Convert the index to strings if necessary.
 - For appending Arrow tables or record batches: Use the first column of the table as RowID. The first column must be of type string.
- "generate": Generate new RowIDs of the format f"Row{i}"

static from_batches(*generator, row_ids: str = 'generate'*)

Create output table where each batch is provided by a generator

Parameters

row_ids – See BatchOutputTable.create.

abstract property num_batches: int

The number of batches written to this output table

Views

knime.scripting.io.view(*obj*) → NodeView

Create an NodeView for the given object.

This method tries to find out the best option to display the given object. First, the method checks if a special view implementation (listed below) exists for the given object. Next, IPython `_repr_html_`, `_repr_svg_`, `_repr_png_`, or `_repr_jpeg_` are used.

Special view implementations:

- HTML: The obj must be of type str and start with “<!DOCTYPE html>”. The document must be self-contained and must not reference external resources. Links to external resources will be opened in an external browser.
- SVG: The obj must be of type str and contain a valid SVG
- PNG: The obj must be of type bytes and contain a PNG image file
- JPEG: The obj must be of type bytes and contain a JPEG image file
- Matplotlib: The obj must be a matplotlib.figure.Figure
- Plotly: The obj must be a plotly.graph_objects.Figure

Parameters

obj – The object which should be displayed

Raises

ValueError – If no view could be created for the given object

knime.scripting.io.view_matplotlib(*fig=None, format='png'*) → NodeView

Create a view showing the given matplotlib figure.

The figure is displayed by exporting it as an SVG. If no figure is given the current active figure is displayed. Note that the figure is closed and should not be used after calling this method.

Parameters

- **fig** – A `matplotlib.figure.Figure` which should be displayed.
- **format** – Format of the view inside the HTML document. Either “png” or “svg”.

Raises

- **ImportError** – If `matplotlib` is not available.
- **TypeError** – If the figure is not a `matplotlib` figure.

`knime.scripting.io.view_seaborn()` → `NodeView`

Create a view showing the current active seaborn figure.

This function just calls `view_matplotlib()` because seaborn plots are just matplotlib figures under the hood.

Raises

ImportError – If `matplotlib` is not available.

`knime.scripting.io.view_plotly(fig)` → `NodeView`

Create a view showing the given plotly figure.

The figure is displayed by exporting it as an HTML document.

To be able to synchronize the selection between the view and other KNIME views the customdata 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])
node_view = view_plotly(fig)
```

Parameters

fig – A `plotly.graph_objects.Figure` object which should be displayed.

Raises

- **ImportError** – If `plotly` is not available.
- **TypeError** – If the figure is not a `plotly` figure.

`knime.scripting.io.view_html(html: str, svg_or_png: str | bytes | None = None, render_fn: Callable[[], str | bytes] | None = None)` → `NodeView`

Create a `NodeView` that displays the given HTML document.

The document must be self-contained and must not reference external resources. Links to external resources will be opened in an external browser.

Parameters

- **html** – A string containing the HTML document.
- **svg_or_png** – A rendered representation of the HTML page. Either a string containing an SVG or a bytes object containing an PNG image
- **render_fn** – A callable that returns an SVG or PNG representation of the page

`knime.scripting.io.view_svg(svg: str)` → `NodeView`

Create a `NodeView` that displays the given SVG.

Parameters

svg – A string containing the SVG.

`knime.scripting.io.view_png(png: bytes) → NodeView`

Create a NodeView that displays the given PNG image.

Parameters

png – The bytes of the PNG image

`knime.scripting.io.view_jpeg(jpeg: bytes) → NodeView`

Create a NodeView that displays the given JPEG image.

Parameters

jpeg – The bytes of the JPEG image

`knime.scripting.io.view_ipy_repr(obj) → NodeView`

Create a NodeView by using the IPython `_repr_*` function of the object.

Tries to use `*_repr_html_*` `*_repr_svg_*` `*_repr_png_*` `*_repr_jpeg_*` in this order.

Parameters

obj – The object which should be displayed

Raises

ValueError – If no view could be created for the given object

class `knime.scripting.io.NodeView(html: str, svg_or_png: str | bytes | None = None, render_fn: Callable[[], str | bytes] | None = None)`

A view of a KNIME node that can be displayed for the user.

Do not create a NodeView directly but use the utility functions `view`, `view_html`, `view_svg`, `view_png`, and `view_jpeg`.

Python Extension Development (Labs)

These classes can be used by developers to implement their own Python nodes for KNIME. For a more detailed description see the [Pure Python Node Extensions Guide](#)

Note: Before KNIME AP 4.7, the module used to access KNIME functionality was called `knime_extension`. This module has been renamed to `knime.extension`.

Nodes

class `knime.extension.PythonNode`

Extend this class to provide a pure Python based node extension to KNIME Analytics Platform.

Users can either use the decorators `@knext.input_table`, `@knext.input_binary`, `@knext.output_table`, `@knext.output_binary`, and `@knext.output_view`, or populate the `input_ports`, `output_ports`, and `output_view` attributes.

Use the Python logging facilities and its `.warning` and `.error` methods to write warnings and errors to the KNIME console. `.info` and `.debug` will only show up in the KNIME console if the log level in KNIME is configured to show these.

Example:

```

import logging
import knime.extension as knext

LOGGER = logging.getLogger(__name__)

category = knext.category("/community", "mycategory", "My Category", "My category_
↳described", icon="icons/category.png")

@knext.node(name="Pure Python Node", node_type=knext.NodeType.LEARNER, icon_path=
↳"icons/icon.png", category=category)
@knext.input_table(name="Input Data", description="We read data from here")
@knext.output_table(name="Output Data", description="Whatever the node has produced
↳")
class TemplateNode(knext.PythonNode):
    # A Python node has a description.

    def configure(self, configure_context, table_schema):
        LOGGER.info(f"Configuring node")
        return table_schema

    def execute(self, exec_context, table):
        return table

```

abstract configure(*config_context: ConfigurationContext, *inputs*)

Configure this Python node.

Parameters

- **config_context** – The ConfigurationContext providing KNIME utilities during execution
- ***inputs** – Each input table spec or binary port spec will be added as parameter, in the same order that the ports were defined.

Returns

Either a single spec, or a tuple or list of specs. The number of specs must match the number of defined output ports, and they must be returned in this order. Alternatively, instead of a spec, a `knext.Column` can be returned (if the spec shall only consist of one column).

Raises

InvalidParametersError – If the current input parameters do not satisfy this node's requirements.

abstract execute(*exec_context: ExecutionContext, *inputs*)

Execute this Python node.

Parameters

- **exec_context** – The ExecutionContext providing KNIME utilities during execution
- ***inputs** – Each input table or binary port object will be added as parameter, in the same order that the ports were defined. Tables will be provided as a `kn.Table`, while binary data will be a plain Python `bytes` object.

Returns

Either a single output object (table or binary), or a tuple or list of objects. The number of output objects must match the number of defined output ports, and they must be returned

in this order. Tables must be provided as a *kn.Table* or *kn.BatchOutputTable*, while binary data should be returned as plain Python *bytes* object.

A node is part of a category:

```
knime.extension.category(path: str, level_id: str, name: str, description: str, icon: str, after: str = "", locked:
                        bool = True)
```

Register a new node category.

A node category must only be created once. Use a string encoding the absolute category path to add nodes to an existing category.

Parameters

- **path** (*Union[str, Category]*) – The absolute “path” that lead to this category e.g. “/io/read”. The segments are the category level-IDs, separated by a slash (“/”). Categories that contain community nodes should be placed in the “/community” category.
- **level_id** (*str*) – The identifier of the level which is used as a path-segment and must be unique at the level specified by “path”.
- **name** (*str*) – The name of this category e.g. “File readers”.
- **description** (*str*) – A short description of the category.
- **icon** (*str*) – File path to 16x16 pixel PNG icon for this category. The path must be relative to the root of the extension.
- **after** (*str, optional*) – Specifies the level-id of the category after which this category should be sorted in. Defaults to “”.
- **locked** (*bool, optional*) – Set this to False to allow extensions from other vendors to add sub-categories or nodes to this category. Defaults to True.

Returns

The full path of the category which can be used to create nodes inside this category.

Return type

str

A node has a type:

```
class knime.extension.NodeType(value)
```

Defines the different node types that are available for Python based nodes.

LEARNER = 'Learner'

A node learning a model that is typically consumed by a PREDICTOR.

MANIPULATOR = 'Manipulator'

A node that manipulates data.

OTHER = 'Other'

A node that doesn't fit one of the other node types.

PREDICTOR = 'Predictor'

A node that predicts something typically using a model provided by a LEARNER.

SINK = 'Sink'

A node consuming data.

SOURCE = 'Source'

A node producing data.

VISUALIZER = 'Visualizer'

A node that visualizes data.

A node's configure method receives a configuration context that lets you interact with KNIME

class `knime.extension.ConfigurationContext(java_config_ctx, flow_variables)`

The ConfigurationContext provides utilities to communicate with KNIME during a node's configure() method.

property `flow_variables: Dict[str, Any]`

The flow variables coming in from KNIME as a dictionary with string keys. The dictionary can be edited and supports flow variables of the following types:

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

set_warning(*message: str*) → None

Sets a warning on the node.

Parameters

message – the warning message to display on the node

A node's execute method receives an execution context that lets you interact with KNIME and e.g. check whether the user has cancelled the execution of your Python node.

class `knime.extension.ExecutionContext(java_ctx, flow_variables)`

The ExecutionContext provides utilities to communicate with KNIME during a node's execute() method.

property `flow_variables: Dict[str, Any]`

The flow variables coming in from KNIME as a dictionary with string keys. The dictionary can be edited and supports flow variables of the following types:

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

is_canceled() → bool

Returns true if this node's execution has been canceled from KNIME. Nodes can check for this property and return early if the execution does not need to finish. Raising a RuntimeError in that case is encouraged.

set_progress(*progress: float, message: str | None = None*)

Set the progress of the execution.

Note that the progress that can be set here is 80% of the total progress of a node execution. The first and last 10% are reserved for data transfer and will be set by the framework.

Parameters

- **progress** – a floating point number between 0.0 and 1.0
- **message** – an optional message to display in KNIME with the progress

set_warning(*message: str*) → None

Sets a warning on the node.

Parameters

- **message** – the warning message to display on the node

Decorators

These decorators can be used to easily configure your Python node.

knime.extension.node(*name: str, node_type: NodeType, icon_path: str, category: str, after: str | None = None, id: str | None = None, is_deprecated: bool = False*) → Callable

Use this decorator to annotate a PythonNode class or function that creates a PythonNode instance that should correspond to a node in KNIME.

knime.extension.input_binary(*name: str, description: str, id: str*)

Use this decorator to define a bytes-serialized port object input of a node.

Parameters

- **name** – The name of the input port
- **description** – A description of the input port.
- **id** – A unique ID identifying the type of the Port. Only Ports with equal ID can be connected in KNIME

knime.extension.input_table(*name: str, description: str*)

Use this decorator to define an input port of type “Table” of a node.

knime.extension.output_binary(*name: str, description: str, id: str*)

Use this decorator to define a bytes-serialized port object output of a node.

Parameters

- **name** –
- **description** –
- **id** – A unique ID identifying the type of the Port. Only Ports with equal ID can be connected in KNIME

knime.extension.output_table(*name: str, description: str*)

Use this decorator to define an output port of type “Table” of a node.

knime.extension.output_view(*name: str, description: str, static_resources: str | None = None*)

Use this decorator to specify that this node produces a view

Parameters

- **name** – The name of the view
- **description** – Description of the view
- **static_resources** – The path to a folder of resources that will be available to the HTML page. The path given here must be relative to the root of the extension. The resources can be accessed by the same relative file path (e.g. “{static_resources}/{filename}”).

Parameters

To add parameterization to your nodes, the configuration dialog can be defined and customized. Each parameter can be used in the nodes execution by accessing `self.param_name`. These parameters can be set up by using the following parameter types. For a more detailed description see [Defining the node’s configuration dialog](#).

```
class knime.extension.IntParameter(label: str | None = None, description: str | None = None, default_value:
    int | Callable[[Version], int] = 0, validator: Callable[[int], None] |
    None = None, min_value: int | None = None, max_value: int | None =
    None, since_version: Version | str | None = None)
```

Parameter class for primitive integer types.

```
class knime.extension.DoubleParameter(label: str | None = None, description: str | None = None,
    default_value: float | Callable[[Version], float] = 0.0, validator:
    Callable[[float], None] | None = None, min_value: float | None =
    None, max_value: float | None = None, since_version: Version | str |
    None = None)
```

Parameter class for primitive float types.

```
class knime.extension.BoolParameter(label: str | None = None, description: str | None = None,
    default_value: bool | Callable[[Version], bool] = False, validator:
    Callable[[bool], None] | None = None, since_version: Version | str |
    None = None)
```

Parameter class for primitive boolean types.

```
class knime.extension.StringParameter(label: str | None = None, description: str | None = None,
    default_value: str | Callable[[Version], str] = "", enum: List[str] |
    None = None, validator: Callable[[str], None] | None = None,
    since_version: Version | str | None = None)
```

Parameter class for primitive string types.

```
class knime.extension.ColumnParameter(label: str | None = None, description: str | None = None,
    port_index: int = 0, column_filter: Callable[[Column], bool] |
    None = None, include_row_key: bool = False,
    include_none_column: bool = False, since_version: str | None =
    None)
```

Parameter class for single columns.

```
class knime.extension.MultiColumnParameter(label: str | None = None, description: str | None = None,
    port_index: int | None = 0, column_filter:
    Callable[[Column], bool] | None = None, since_version:
    Version | str | None = None)
```

Parameter class for multiple columns.

```
class knime.extension.EnumParameter(label: str | None = None, description: str | None = None,
    default_value: str | Callable[[Version], str] | None = None, enum:
    EnumParameterOptions | None = None, validator: Callable[[str],
    None] | None = None, since_version: Version | str | None = None)
```


Parameter class for multiple-choice parameter types. Replicates and extends the enum functionality previously implemented as part of `StringParameter`.

A subclass of `EnumParameterOptions` should be provided as the enum parameter, which should contain class attributes of the form `OPTION_NAME = (OPTION_LABEL, OPTION_DESCRIPTION)`. The corresponding option attributes can be accessed via `MyOptions.OPTION_NAME.name`, `.label`, and `.description` respectively.

The `.name` attribute of each option is used as the selection constant, e.g. `MyOptions.OPTION_NAME.name == "OPTION_NAME"`.

Example:

```
class CoffeeOptions(EnumParameterOptions):
    CLASSIC = ("Classic", "The classic chocolatey taste, with notes of bitterness_
    ↪and wood.")
    FRUITY = ("Fruity", "A fruity taste, with notes of berries and citrus.")
    WATERY = ("Watery", "A watery taste, with notes of water and wetness.")

    coffee_selection_param = knext.EnumParameter(
        label="Coffee Selection",
        description="Select the type of coffee you like to drink.",
        default_value=CoffeeOptions.CLASSIC.name,
        enum=CoffeeOptions,
    )
```

class `knime.extension.EnumParameterOptions(value)`

A helper class for creating `EnumParameterOptions`, based on Python's `Enum` class.

Developers should subclass this class, and provide enumeration options as class attributes of the subclass, of the form `OPTION_NAME = (OPTION_LABEL, OPTION_DESCRIPTION)`.

Enum option objects can be accessed as attributes of the `EnumParameterOptions` subclass, e.g. `MyEnum.OPTION_NAME`. Each option object has the following attributes:

- `name`: the name of the class attribute, e.g. `"OPTION_NAME"`, which is used as the selection constant;
- `label`: the label of the option, displayed in the configuration dialogue of the node;
- `description`: the description of the option, used along with the label to generate a list of the available options in the Node Description and in the configuration dialogue of the node.

Example:

```
class CoffeeOptions(EnumParameterOptions):
    CLASSIC = ("Classic", "The classic chocolatey taste, with notes of bitterness_
    ↪and wood.")
    FRUITY = ("Fruity", "A fruity taste, with notes of berries and citrus.")
    WATERY = ("Watery", "A watery taste, with notes of water and wetness.")
```

classmethod `get_all_options()`

Returns a list of all options defined in the `EnumParameterOptions` subclass.

Validation

While each parameter type listed above has default type validation (eg checking if the `IntParameter` contains only Integers), they also support custom validation via a property-like decorator notation. For instance, this can be used to verify that the parameter value matches a certain criteria (see example below). The validator should be placed below the definition of the corresponding parameter.

```
class knime.extension.IntParameter(label: str | None = None, description: str | None = None, default_value:
    int | Callable[[Version], int] = 0, validator: Callable[[int], None] |
    None = None, min_value: int | None = None, max_value: int | None =
    None, since_version: Version | str | None = None)
```

Parameter class for primitive integer types.

validator(func)

To be used as a decorator for setting a validator function for a parameter. Note that ‘func’ will be encapsulated in ‘_validator’ and will not be available in the namespace of the class.

Example:

```
@knext.node(args)
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(args):
        pass

    def execute(args):
        pass
```

Parameter Groups

Additionally these parameters can be combined in `parameter_groups`. These groups are visualized as sections in the configuration dialog. 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.

```
knime.extension.parameter_group(label: str, since_version: Version | str | None = None)
```

Used for injecting descriptor protocol methods into a custom parameter group class. “obj” in this context is the parameterized object instance or a parameter group instance.

Group validators need to raise an exception if a values-based condition is violated, where values is a dictionary of parameter names and values. Group validators can be set using either of the following methods:

- By implementing the “validate(self, values)” method inside the class definition of the group.

Example:

```
def validate(self, values):
    assert values['first_param'] + values['second_param'] < 100
```

- By using the “@group_name.validator” decorator notation inside the class definition of the “parent” of the group. The decorator has an optional ‘override’ parameter, set to True by default, which overrides the “validate” method. If ‘override’ is set to False, the “validate” method, if defined, will be called first.

Example:

```
@hyperparameters.validator(override=False)
def validate_hyperparams(values):
    assert values['first_param'] + values['second_param'] < 100
```

Example:

```
@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(args)
    class MyNodeWithSettings:
        settings = MySettings()
        def configure(args):
            pass

        def execute(args):
            pass
```

Tables

Table and Schema are the two classes that are used to communicate tabular data (Table) during execute, or the table structure (Schema) in configure between Python and KNIME.

class knime.extension.Table

This class serves as public API to create KNIME tables either from pandas or pyarrow. These tables can then be sent back to KNIME. This class has to be instantiated by calling either `from_pyarrow()` or `from_pandas()`

__getitem__(*slicing: slice | List[int] | List[str] | Tuple[slice | List[int] | List[str], slice]*) → `_TabularView`

Creates a view of this Table by slicing rows and columns. The slicing syntax is similar to that of numpy arrays, but columns can also be addressed as index lists or via a list of column names.

The syntax is `[column_slice, row_slice]`. Note that this is the exact opposite order than in the deprecated scripting API's `ReadTable`.

Parameters

- **column_slice** – A column index, a column name, a slice object, a list of column indices, or a list of column names.
- **row_slice** – Optional: A slice object describing which rows to use.

Returns

A `_TabularView` representing a slice of the original Table

Example:

```
row_sliced_table = table[:, :100] # Get the first 100 rows
column_sliced_table = table[["name", "age"]] # Get all rows of the columns
→ "name" and "age"
row_and_column_sliced_table = table[1:5, :100] # Get the first 100 rows of
→ columns 1,2,3,4
```

batches() → Iterator[Table]

Returns a generator over the batches in this table. A batch is part of the table with all columns, but only a subset of the rows. A batch should always fit into memory (max size currently 64mb). The table being passed to `execute()` is already present in batches, so accessing the data this way is very efficient.

Example:

```
output_table = BatchOutputTable.create()
for batch in my_table.batches():
    input_batch = batch.to_pandas()
    # process the batch
    output_table.append(Table.from_pandas(input_batch))
```

static from_pandas(data: pandas.DataFrame, sentinel: str | int | None = None, row_ids: str = 'auto')

Factory method to create a Table given a pandas.DataFrame. The index of the data frame will be used as RowKey by KNIME.

Example:

```
Table.from_pandas(my_pandas_df, sentinel="min")
```

Parameters

- **data** – A pandas.DataFrame
- **sentinel** – Interpret the following values in integral columns as missing value:
 - "min" min int32 or min int64 depending on the type of the column
 - "max" max int32 or max int64 depending on the type of the column
 - a special integer value that should be interpreted as missing value
- **row_ids** – Defines what RowID should be used. Must be one of the following values:
 - "keep": Keep the DataFrame.index as the RowID. Convert the index to strings if necessary.
 - "generate": Generate new RowIDs of the format f"Row{i}" where i is the position of the row (from 0 to length-1).
 - "auto": If the DataFrame.index is of type int or unsigned int, use f"Row{n}" where n is the index of the row. Else, use "keep".

static from_pyarrow(data: pyarrow.Table, sentinel: str | int | None = None, row_ids: str = 'auto')

Factory method to create a Table given a pyarrow.Table.

Example:

```
Table.from_pyarrow(my_pyarrow_table, sentinel="min")
```

Parameters

- **data** – A `pyarrow.Table`
- **sentinel** – Interpret the following values in integral columns as missing value:
 - "min" min int32 or min int64 depending on the type of the column
 - "max" max int32 or max int64 depending on the type of the column
 - a special integer value that should be interpreted as missing value
- **row_ids** – Defines what RowID should be used. Must be one of the following values:
 - "keep": Use the first column of the table as RowID. The first column must be of type string.
 - "generate": Generate new RowIDs of the format `f"Row{i}"` where `i` is the position of the row (from 0 to `length-1`).
 - "auto": Use the first column of the table if it has the name "<RowID>" and is of type string or integer.
 - * If the "<RowID>" column is of type string, use it directly
 - * If the "<RowID>" column is of an integer type use `f"Row{n}"` where `n` is the value of the integer column.
 - * Generate new RowIDs ("generate") if the first column has another type or name.

remove(*slicing: str | int | List[str]*)

Implements remove method for Columnar data structures. The input can be a column index, a column name or a list of column names.

If the input is a column index, the column with that index will be removed. If it is a column name, then the first column with matching name is removed. Passing a list of column names will filter out all (including duplicate) columns with matching names.

Parameters

slicing – Can be of type integer representing the index in `column_names` to remove. Or a list of strings removing every column matching from that list. Or a string of which first occurrence is removed from the `column_names`.

Returns

A View missing the columns to be removed.

Raises

- **ValueError** if no matching column is found given a list or str –
- **IndexError** if column is accessed by integer and is out of bounds –
- **TypeError** if the key is neither a integer nor a string or list of strings. –

abstract property schema: `Schema`

The schema of this table, containing column names, types, and potentially metadata

to_batches() → `Iterator[Table]`

Alias for `Table.batches()`

to_pandas(*sentinel: str | int | None = None*) → `pandas.DataFrame`

Access this table as a `pandas.DataFrame`.

Parameters

sentinel – Replace missing values in integral columns by the given value, one of:

- "min" min int32 or min int64 depending on the type of the column
- "max" max int32 or max int64 depending on the type of the column
- An integer value that should be inserted for each missing value

to_pyarrow(*sentinel: str | int | None = None*) → `pyarrow.Table`

Access this table as a `pyarrow.Table`.

Parameters

sentinel – Replace missing values in integral columns by the given value, one of:

- "min" min int32 or min int64 depending on the type of the column
- "max" max int32 or max int64 depending on the type of the column
- An integer value that should be inserted for each missing value

class knime.extension.BatchOutputTable

An output table generated by combining smaller tables (also called batches).

All batches must have the same number, names and types of columns.

Does not provide means to continue to work with the data but is meant to be used as a return value of a Node's `execute()` method.

abstract append(*batch: Table | pandas.DataFrame | pyarrow.Table | pyarrow.RecordBatch*) → `None`

Append a batch to this output table. The first batch defines the structure of the table, and all subsequent batches must have the same number of columns, column names and column types.

Note: Keep in mind that the RowID will be handled according to the "row_ids" mode chosen in `BatchOutputTable.create`.

static create(*row_ids: str = 'keep'*)

Create an empty `BatchOutputTable`

Parameters

row_ids – Defines what RowID should be used. Must be one of the following values:

- "keep":
 - For appending DataFrames: Keep the `DataFrame.index` as the RowID. Convert the index to strings if necessary.
 - For appending Arrow tables or record batches: Use the first column of the table as RowID. The first column must be of type string.
- "generate": Generate new RowIDs of the format `f"Row{i}"`

static from_batches(*generator, row_ids: str = 'generate'*)

Create output table where each batch is provided by a generator

Parameters

row_ids – See `BatchOutputTable.create`.

abstract property num_batches: int

The number of batches written to this output table

```
class knime.extension.Schema(ktypes: List[KnimeType | Type], names: List[str], metadata: List | None = None)
```

A schema defines the data types and names of the columns inside a table. Additionally, it can hold metadata for the individual columns.

```
__getitem__(slicing: slice | List[int] | List[str]) → _ColumnarView
```

Creates a view of this Table or Schema by slicing columns. The slicing syntax is similar to that of numpy arrays, but columns can also be addressed as index lists or via a list of column names.

Parameters

column_slice – A column index, a column name, a slice object, a list of column indices, or a list of column names. For single indices, the view will create a “Column” object. For slices or lists of indices, a new Schema will be returned.

Returns

A _ColumnarView representing a slice of the original Schema or Table.

Examples:

Get columns 1,2,3,4: `sliced_schema = schema[1:5]`

Get the columns “name” and “age”: `sliced_schema = schema[["name", "age"]]`

```
property column_names: List[str]
```

Return the list of column names

```
classmethod deserialize(table_schema: dict) → Schema
```

Construct a Schema from a dict that was retrieved from KNIME in JSON encoded form as the input to a node’s configure() method.

KNIME provides table information with a RowKey column at the beginning, which we drop before returning the created schema.

```
classmethod from_columns(columns: Sequence[Column] | Column)
```

Create a schema from a single column or a list of columns

```
classmethod from_types(ktypes: List[KnimeType | Type], names: List[str], metadata: List | None = None)
```

Create a schema from a list of column data types, names and metadata

```
property num_columns
```

The number of columns in this schema

```
remove(slicing: str | int | List[str])
```

Implements remove method for Columnar data structures. The input can be a column index, a column name or a list of column names.

If the input is a column index, the column with that index will be removed. If it is a column name, then the first column with matching name is removed. Passing a list of column names will filter out all (including duplicate) columns with matching names.

Parameters

slicing – Can be of type integer representing the index in column_names to remove. Or a list of strings removing every column matching from that list. Or a string of which first occurrence is removed from the column_names.

Returns

A View missing the columns to be removed.

Raises

- **ValueError** if no matching column is found given a list or str –

- **IndexError** if `column` is accessed by integer and is out of bounds –
- **TypeError** if the key is neither a integer nor a string or list of strings. –

serialize() → Dict

Convert this Schema into dict which can then be JSON encoded and sent to KNIME as result of a node's `configure()` method.

Because KNIME expects a row key column as first column of the schema, but we don't include this in the KNIME Python table schema, we insert a row key column here.

Raises

RuntimeError – if duplicate column names are detected

class `knime.extension.Column(ktype: KnimeType | Type, name: str, metadata=None)`

A column inside a table schema consists of the knime datatype, a column name and optional metadata.

__init__(*ktype: KnimeType | Type, name: str, metadata=None*)

Construct a Column from type, name and optional metadata.

Parameters

- **ktype** – The KNIME type of the column or a type which can be converted via `knime.api.schema.logical(ktype)` to a KNIME type
- **name** – The name of the column. May not be empty.

Raises

- **TypeError** – if the type is no KNIME type or cannot be converted to a KNIME type
- **ValueError** – if the name is empty

Data Types

These are helper functions to create KNIME compatible datatypes. For instance, if a new column is created.

`knime.extension.int32()`

Create a KNIME integer type with 32 bits

`knime.extension.int64()`

Create a KNIME integer type with 64 bits

`knime.extension.double()`

Create a KNIME floating point type with double precision (64 bits)

`knime.extension.bool_()`

Create a KNIME boolean type

`knime.extension.string(dict_encoding_key_type: DictEncodingKeyType | None = None)`

Create a KNIME string type.

Parameters

dict_encoding_key_type – The key type to use for dictionary encoding. If this is None (the default), no dictionary encoding will be used. Dictionary encoding helps to reduce storage space and read/write performance for columns with repeating values such as categorical data.

`knime.extension.blob(dict_encoding_key_type: DictEncodingKeyType | None = None)`

Create a KNIME blob type for binary data of variable length

Parameters

dict_encoding_key_type – The key type to use for dictionary encoding. If this is None (the default), no dictionary encoding will be used. Dictionary encoding helps to reduce storage space and read/write performance for columns with repeating values such as categorical data.

`knime.extension.list_(inner_type: KnimeType)`

Create a KNIME type that is a list of the given inner types

Parameters

inner_type – The type of the elements in the list. Must be a KnimeType

`knime.extension.struct(*inner_types)`

Create a KNIME structured data type where each given argument represents a field of the struct.

Parameters

inner_types – The argument list of this method defines the fields in this structured data type. Each inner type must be a KNIME type

`knime.extension.logical(value_type) → LogicalType`

Create a KNIME logical data type of the given Python value type.

Parameters

value_type – The type of the values inside this column. A `knime.api.types.PythonValueFactory` must be registered for this type.

Raises

TypeError – if no `PythonValueFactory` has been registered for this value type with `knime.api.types.register_python_value_factory`

Deprecated Python Script API

This section lists the API of the module `knime_io` that functioned as the main contact point between KNIME and Python in the [KNIME Python Script node](#) in KNIME AP before version 4.7, when the Python Script node was moved out of Labs. Please refer to the [KNIME Python Integration Guide](#) for more details on how to set up and use the node.

Warning: This API is deprecated since KNIME AP 4.7, please use the current API as described in [Python Script API](#)

Inputs and outputs

These properties can be used to retrieve data from or pass data back to KNIME Analytics Platform. The length of the input and output lists depends on the number of input and output ports of the node.

Example: If you have a Python Script node configured with two input tables and one input object, you can access the two tables via `knime_io.input_tables[0]` and `knime_io.input_tables[1]`, and the input object via `knime_io.input_objects[0]`.

`knime_io.flow_variables: Dict[str, Any] = {}`

A dictionary of flow variables provided by the KNIME workflow. New flow variables can be added to the output of the node by adding them to the dictionary. Supported flow variable types are numbers, strings, booleans and lists thereof.

knime_io.input_objects: List = <knime.scripting.io_containers._FixedSizeListView object>

A list of input objects of this script node using zero-based indices. This list has a fixed size, which is determined by the number of input object ports configured for this node. Input objects are Python objects that are passed in from another Python script node's ``output_object`` port. This can, for instance, be used to pass trained models between Python nodes. If no input is given, the list exists but is empty.

knime_io.input_tables: List[ReadTable] = <knime.scripting.io_containers._FixedSizeListView object>

The input tables of this script node. This list has a fixed size, which is determined by the number of input table ports configured for this node. Tables are available in the same order as the port connectors are displayed alongside the node (from top to bottom), using zero-based indexing. If no input is given, the list exists but is empty.

knime_io.output_images: List = <knime.scripting.io_containers._FixedSizeListView object>

The output images of this script node. This list has a fixed size, which is determined by the number of output images configured for this node. The value passed to the output port should be an array of bytes encoding an SVG or PNG image.

Example:

```
data = knime_io.input_tables[0].to_pandas()
buffer = io.BytesIO()

pyplot.figure()
pyplot.plot('x', 'y', data=data)
pyplot.savefig(buffer, format='svg')

knime_io.output_images[0] = buffer.getvalue()
```

knime_io.output_objects: List = <knime.scripting.io_containers._FixedSizeListView object>

The output objects of this script node. This list has a fixed size, which is determined by the number of output object ports configured for this node. Each output object can be an arbitrary Python object as long as it can be *pickled*. Use this to, for example, pass a trained model to another Python script node.

Example:

```
model = torchvision.models.resnet18()
...
# train/finetune model
...
knime_io.output_objects[0] = model
```

knime_io.output_tables: List[WriteTable] = <knime.scripting.io_containers._FixedSizeListView object>

The output tables of this script node. This list has a fixed size, which is determined by the number of output table ports configured for this node. You should assign a WriteTable or BatchWriteTable to each output port of this node. See the factory methods `knime_io.write_table()` and `knime_io.batch_write_table()` below.

Example:

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

Factory methods

Use these methods to fill the `knime_io.output_tables`.

`knime_io.batch_write_table()` → `BatchWriteTable`

Factory method to create an empty `BatchWriteTable` that can be filled sequentially batch by batch (see Example).

Example:

```
table = knime_io.batch_write_table()
table.append(df_1)
table.append(df_2)
knime_io.output_tables[0] = table
```

Warning: This class is deprecated since KNIME AP 4.7, use `knime.api.table.BatchOutputTable.create()` instead.

`knime_io.write_table(data: ReadTable | pandas.DataFrame | pyarrow.Table, sentinel: str | int | None = None)`
→ `WriteTable`

Factory method to create a `WriteTable` given a `pandas.DataFrame` or a `pyarrow.Table`. If the input is a `pyarrow.Table`, its first column must contain unique row identifiers of type 'string'.

Example:

```
knime_io.output_tables[0] = knime_io.write_table(my_pandas_df, sentinel="min")
```

Parameters

- **data** – A `ReadTable`, `pandas.DataFrame` or a `pyarrow.Table`
- **sentinel** – Interpret the following values in integral columns as missing value:
 - "min" min int32 or min int64 depending on the type of the column
 - "max" max int32 or max int64 depending on the type of the column
 - a special integer value that should be interpreted as missing value

Warning: This method is deprecated since KNIME AP 4.7, use `knime.api.table.Table.from_pandas()` or `knime.api.table.Table.from_pyarrow()` instead.

Classes

class `knime.scripting._deprecated._table.Batch`

A batch is a part of a table containing data. A batch should always fit into system memory, thus all methods accessing the data will be processed immediately and synchronously.

It can be sliced before the data is accessed as `pandas.DataFrame` or `pyarrow.RecordBatch`.

`__getitem__` (slicing: `slice` | `Tuple[slice, slice]` | `List[int]` | `List[str]`) → `SlicedDataView`

Creates a view of this batch by slicing specific rows and columns. The slicing syntax is similar to that of numpy arrays, but columns can also be addressed as index lists or via a list of column names.

Parameters

- **row_slice** – A slice object describing which rows to use.
- **column_slice** – Optional. A slice object, a list of column indices, or a list of column names.

Returns

A SlicedDataView that can be converted to pandas or pyarrow.

Example:

```
full_batch = batch[:] # Slice/Get the full batch

# Slicing works for rows and columns. Column slices can be defined with int's,
→ or the column names
row_sliced_batch = batch[:100] # Get first 100 rows of the batch
column_sliced_batch = batch[:, ["name", "age"]] # Get all rows of the columns
→ "name" and "age"
row_and_column_sliced_batch = batch[:100, 1:5] # Get the first 100 rows of
→ columns 1,2,3,4

# The resulting `sliced_batches` cannot be sliced further. But they can be
→ converted to pandas or pyarrow.
```

abstract property column_names: Tuple[str, ...]

Returns the list of column names.

abstract property num_columns: int

Returns the number of columns in the table.

abstract property num_rows: int

Returns the number of rows in the table.

If the table is not completely available yet because batches are still appended to it, querying the number of rows blocks until all data is available.

property shape: Tuple[int, int]

Returns a tuple in the form (numRows, numColumns) representing the shape of this table.

If the table is not completely available yet because batches are still appended to it, querying the shape blocks until all data is available.

abstract to_pandas(sentinel: str | int | None = None) → pandas.DataFrame

Access the batch or table as a pandas.DataFrame.

Parameters

sentinel – Replace missing values in integral columns by the given value, one of:

- "min" min int32 or min int64 depending on the type of the column
- "max" max int32 or max int64 depending on the type of the column
- An integer value that should be inserted for each missing value

Raises

IndexError – If rows or columns were requested outside of the available shape

abstract to_pyarrow(sentinel: str | int | None = None) → pyarrow.RecordBatch | pyarrow.Table

Access this batch or table as a pyarrow.RecordBatch or pyarrow.table. The returned type depends on the type of the underlying object. When called on a ReadTable, returns a pyarrow.Table.

Parameters

sentinel – Replace missing values in integral columns by the given value, one of:

- "min" min int32 or min int64 depending on the type of the column
- "max" max int32 or max int64 depending on the type of the column
- An integer value that should be inserted for each missing value

Raises

IndexError – If rows or columns were requested outside of the available shape

class knime.scripting._deprecated._table.**ReadTable**

A KNIME ReadTable provides access to the data provided from KNIME, either in full (must fit into memory) or split into row-wise batches.

Warning: This class is deprecated since KNIME AP 4.7, use `knime.api.table.Table` instead.

__getitem__(*slicing: slice | Tuple[slice, slice | List[int] | List[str]]*) → SlicedDataView

Creates a view of this ReadTable by slicing rows and columns. The slicing syntax is similar to that of numpy arrays, but columns can also be addressed as index lists or via a list of column names.

The returned *sliced_table* cannot be sliced further. But they can be converted to pandas or pyarrow.

Parameters

- **row_slice** – A slice object describing which rows to use.
- **column_slice** – Optional. A slice object, a list of column indices, or a list of column names.

Returns

a SlicedDataView that can be converted to pandas or pyarrow.

Example:

```
row_sliced_table = table[:100] # Get the first 100 rows
column_sliced_table = table[:, ["name", "age"]] # Get all rows of the columns
→ "name" and "age"
row_and_column_sliced_table = table[:100, 1:5] # Get the first 100 rows of
→ columns 1,2,3,4

df = row_and_column_sliced_table.to_pandas()
```

__len__() → int

Returns the number of batches of this table

abstract batches() → Iterator[Batch]

Returns an generator for the batches in this table. If the generator is advanced to a batch that is not available yet, it will block until the data is present. `len(my_read_table)` gives the static amount of batches within the table, which is not updated.

Example:

```
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)
```

abstract property column_names: `Tuple[str, ...]`

Returns the list of column names.

abstract property num_batches: `int`

Returns the number of batches in this table.

If the table is not completely available yet because batches are still appended to it, querying the number of batches blocks until all data is available.

abstract property num_columns: `int`

Returns the number of columns in the table.

abstract property num_rows: `int`

Returns the number of rows in the table.

If the table is not completely available yet because batches are still appended to it, querying the number of rows blocks until all data is available.

property shape: `Tuple[int, int]`

Returns a tuple in the form (numRows, numColumns) representing the shape of this table.

If the table is not completely available yet because batches are still appended to it, querying the shape blocks until all data is available.

abstract to_pandas(*sentinel*: `str | int | None = None`) \rightarrow `pandas.DataFrame`

Access the batch or table as a `pandas.DataFrame`.

Parameters

sentinel – Replace missing values in integral columns by the given value, one of:

- "min" min int32 or min int64 depending on the type of the column
- "max" max int32 or max int64 depending on the type of the column
- An integer value that should be inserted for each missing value

Raises

IndexError – If rows or columns were requested outside of the available shape

abstract to_pyarrow(*sentinel*: `str | int | None = None`) \rightarrow `pyarrow.RecordBatch | pyarrow.Table`

Access this batch or table as a `pyarrow.RecordBatch` or `pyarrow.table`. The returned type depends on the type of the underlying object. When called on a `ReadTable`, returns a `pyarrow.Table`.

Parameters

sentinel – Replace missing values in integral columns by the given value, one of:

- "min" min int32 or min int64 depending on the type of the column
- "max" max int32 or max int64 depending on the type of the column
- An integer value that should be inserted for each missing value

Raises

IndexError – If rows or columns were requested outside of the available shape

class `knime.scripting._deprecated._table.WriteTable`

A table that can be filled as a whole.

Warning: This class is deprecated since KNIME AP 4.7, use <code>knime.api.table.Table</code> instead.
--

abstract property column_names: `Tuple[str, ...]`

Returns the list of column names.

abstract property num_batches: `int`

Returns the number of batches in this table.

If the table is not completely available yet because batches are still appended to it, querying the number of batches blocks until all data is available.

abstract property num_columns: `int`

Returns the number of columns in the table.

abstract property num_rows: `int`

Returns the number of rows in the table.

If the table is not completely available yet because batches are still appended to it, querying the number of rows blocks until all data is available.

property shape: `Tuple[int, int]`

Returns a tuple in the form (numRows, numColumns) representing the shape of this table.

If the table is not completely available yet because batches are still appended to it, querying the shape blocks until all data is available.

class `knime.scripting._deprecated._table.BatchWriteTable`

A table that can be filled batch by batch.

Warning: This class is deprecated since KNIME AP 4.7, use `knime.api.table.BatchOutputTable` instead.

abstract append(*data: Batch | pandas.DataFrame | pyarrow.RecordBatch, sentinel: str | int | None = None*)

Appends a batch with the given data to the end of this table. The number of columns, as well as their data types, must match that of the previous batches in this table. Note that this cannot take a `pyarrow.Table` as input. With `pyarrow`, it can only process batches, which can be created as follows from some input table.

Example:

```
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)
```

Parameters

- **data** – A batch, a `pandas.DataFrame` or a `pyarrow.RecordBatch`
- **sentinel** – Only if data is a `pandas.DataFrame` or `pyarrow.RecordBatch`. Interpret the following values in integral columns as missing value:
 - "min" min int32 or min int64 depending on the type of the column
 - "max" max int32 or max int64 depending on the type of the column
 - a special integer value that should be interpreted as missing value

Raises

ValueError – If the new batch does not have the same columns as previous batches in this Writetable.

abstract property column_names: `Tuple[str, ...]`

Returns the list of column names.

static create() \rightarrow `BatchWriteTable`

Create an empty `BatchWriteTable`

abstract property num_batches: `int`

Returns the number of batches in this table.

If the table is not completely available yet because batches are still appended to it, querying the number of batches blocks until all data is available.

abstract property num_columns: `int`

Returns the number of columns in the table.

abstract property num_rows: `int`

Returns the number of rows in the table.

If the table is not completely available yet because batches are still appended to it, querying the number of rows blocks until all data is available.

property shape: `Tuple[int, int]`

Returns a tuple in the form (numRows, numColumns) representing the shape of this table.

If the table is not completely available yet because batches are still appended to it, querying the shape blocks until all data is available.

License

Please see below the General Public License (GPL), Version 3,
and the Additional Permissions according to Sec. 7
applying to the files in this folder:

*** **

GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <<http://fsf.org/>>

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for
software and other kinds of works.

The licenses for most software and other practical works are designed
to take away your freedom to share and change the works. By contrast,
the GNU General Public License is intended to guarantee your freedom to
share and change all versions of a program--to make sure it remains free

software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of

packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your

work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

- a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
- b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
- c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
- d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for

unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

- a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
- b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
- c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
- d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
- e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
- f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms

of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically

receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means,

then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single

combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>  
Copyright (C) <year> <name of author>
```

```
This program is free software: you can redistribute it and/or modify  
it under the terms of the GNU General Public License as published by  
the Free Software Foundation, either version 3 of the License, or  
(at your option) any later version.
```

```
This program is distributed in the hope that it will be useful,  
but WITHOUT ANY WARRANTY; without even the implied warranty of  
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the  
GNU General Public License for more details.
```

```
You should have received a copy of the GNU General Public License  
along with this program. If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program> Copyright (C) <year> <name of author>  
This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.  
This is free software, and you are welcome to redistribute it  
under certain conditions; type `show c' for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also give your employer (if you work as a programmer) or school,

if any, to sign a "copyright disclaimer" for the program, if necessary.
 For more information on this, and how to apply and follow the GNU GPL, see
[<http://www.gnu.org/licenses/>](http://www.gnu.org/licenses/).

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read
[<http://www.gnu.org/philosophy/why-not-lgpl.html>](http://www.gnu.org/philosophy/why-not-lgpl.html).

*** **

Additional permissions under GNU GPL Version 3 Section 7:

KNIME interoperates with ECLIPSE solely via ECLIPSE's plug-in APIs. Hence, KNIME and ECLIPSE are both independent programs and are not derived from each other. Should, however, the interpretation of the GNU GPL Version 3 ("License") under any applicable laws result in KNIME and ECLIPSE being a combined program, KNIME AG herewith grants you the additional permission to use and propagate KNIME together with ECLIPSE with only the license terms in place for ECLIPSE applying to ECLIPSE and the GNU GPL Version 3 applying for KNIME, provided the license terms of ECLIPSE themselves allow for the respective use and propagation of ECLIPSE together with KNIME.

Additional permission relating to nodes for KNIME that extend the Node Extension (and in particular that are based on subclasses of NodeModel, NodeDialog, and NodeView) and that only interoperate with KNIME through standard APIs ("Nodes"):

Nodes are deemed to be separate and independent programs and to not be covered works. Notwithstanding anything to the contrary in the License, the License does not apply to Nodes, you are not required to license Nodes under the License, and you are granted a license to prepare and propagate Nodes, in each case even if such Nodes are propagated with or for interoperation with KNIME. The owner of a Node may freely choose the license terms applicable to such Node, including when such Node is propagated with or for interoperation with KNIME.