KNIME Python API

Carsten Haubold, Adrian Nembach, Marcel Wiedenmann, Benjami

Jan 31, 2023

CONTENTS

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

CHAPTER ONE

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 than 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 occurence 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 fuction 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 than 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 occurence 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 occurence 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 than 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 occurence 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

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 fuction 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 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␣
↪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:

```python
@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:

```python
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 than 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 occurence 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 occurence 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*) → 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.

License