

# Neues Python-basiertes KNIME erstellen

## Erweiterung

KNIME AG, Zürich, Schweiz

Version 5.7 (letzte Aktualisierung auf )



## Inhaltsverzeichnis

<a href="#page3" style="color: #000000; text-decoration: underline;">Einleitung . . . . .</a>	<a href="#page3" style="color: #000000; text-decoration: underline;">1</a>
<a href="#page4" style="color: #000000; text-decoration: underline;">Quickstart Tutorial . . .</a>	<a href="#page4" style="color: #000000; text-decoration: underline;">2</a>
<a href="#page4" style="color: #000000; text-decoration: underline;">Voraussetzungen .</a>	<a href="#page4" style="color: #000000; text-decoration: underline;">3</a>
<a href="#page6" style="color: #000000; text-decoration: underline;">Schreiben Sie Ihren ersten Python-Code</a>	<a href="#page6" style="color: #000000; text-decoration: underline;">4</a>
<a href="#page11" style="color: #000000; text-decoration: underline;">Python Node Extension Setup</a>	<a href="#page11" style="color: #000000; text-decoration: underline;">11</a>
<a href="#page12" style="color: #000000; text-decoration: underline;">Entwicklung und Verteilung</a>	<a href="#page12" style="color: #000000; text-decoration: underline;">12</a>
<a href="#page13" style="color: #000000; text-decoration: underline;">Definieren einer KNIME Node</a>	<a href="#page13" style="color: #000000; text-decoration: underline;">13</a>
<a href="#page14" style="color: #000000; text-decoration: underline;">Definieren von benutzerspezifischen Parametern</a>	<a href="#page14" style="color: #000000; text-decoration: underline;">14</a>
<a href="#page16" style="color: #000000; text-decoration: underline;">Keine Portkonfiguration</a>	<a href="#page16" style="color: #000000; text-decoration: underline;">16</a>
<a href="#page20" style="color: #000000; text-decoration: underline;">Angabe der Knotenkategorie</a>	<a href="#page20" style="color: #000000; text-decoration: underline;">20</a>
<a href="#page25" style="color: #000000; text-decoration: underline;">Definieren des Konfigurationsdialogs</a>	<a href="#page25" style="color: #000000; text-decoration: underline;">25</a>
<a href="#page33" style="color: #000000; text-decoration: underline;">Keine Ansichtserklärung</a>	<a href="#page33" style="color: #000000; text-decoration: underline;">33</a>
<a href="#page34" style="color: #000000; text-decoration: underline;">Zugriff auf Durchflussobjekte</a>	<a href="#page34" style="color: #000000; text-decoration: underline;">34</a>
<a href="#page35" style="color: #000000; text-decoration: underline;">Ausführung Ihrer Extension</a>	<a href="#page35" style="color: #000000; text-decoration: underline;">35</a>
<a href="#page39" style="color: #000000; text-decoration: underline;">Definition von Knoten</a>	<a href="#page39" style="color: #000000; text-decoration: underline;">39</a>
<a href="#page40" style="color: #000000; text-decoration: underline;">Verbesserung der Knotenbasis</a>	<a href="#page40" style="color: #000000; text-decoration: underline;">40</a>
<a href="#page45" style="color: #000000; text-decoration: underline;">Teile deine Extension</a>	<a href="#page45" style="color: #000000; text-decoration: underline;">45</a>
<a href="#page45" style="color: #000000; text-decoration: underline;">Einrichtung. . . .</a>	<a href="#page45" style="color: #000000; text-decoration: underline;">45</a>
<a href="#page48" style="color: #000000; text-decoration: underline;">Option 1: Bündeln in eine Python-Datei</a>	<a href="#page48" style="color: #000000; text-decoration: underline;">48</a>
<a href="#page49" style="color: #000000; text-decoration: underline;">Option 2: Veröffentlichung</a>	<a href="#page49" style="color: #000000; text-decoration: underline;">49</a>
<a href="#page52" style="color: #000000; text-decoration: underline;">Anpassen der Python-Code</a>	<a href="#page52" style="color: #000000; text-decoration: underline;">52</a>
<a href="#page53" style="color: #000000; text-decoration: underline;">Registrierung von Python-Modulen</a>	<a href="#page53" style="color: #000000; text-decoration: underline;">53</a>
<a href="#page54" style="color: #000000; text-decoration: underline;">Andere Themen . . .</a>	<a href="#page54" style="color: #000000; text-decoration: underline;">54</a>
<a href="#page54" style="color: #000000; text-decoration: underline;">Loggen. . . . .</a>	<a href="#page54" style="color: #000000; text-decoration: underline;">54</a>
<a href="#page54" style="color: #000000; text-decoration: underline;">Gateway-Cache.</a>	<a href="#page54" style="color: #000000; text-decoration: underline;">54</a>
<a href="#page56" style="color: #000000; text-decoration: underline;">Fehlerbehebung . .</a>	<a href="#page56" style="color: #000000; text-decoration: underline;">56</a>
<a href="#page56" style="color: #000000; text-decoration: underline;">Finde Debug-Informationen</a>	<a href="#page56" style="color: #000000; text-decoration: underline;">56</a>
<a href="#page57" style="color: #000000; text-decoration: underline;">Wie man Python-Veränderungen aktiviert</a>	<a href="#page57" style="color: #000000; text-decoration: underline;">57</a>
<a href="#page57" style="color: #000000; text-decoration: underline;">Entwickeln Sie mehr Python-Code</a>	<a href="#page57" style="color: #000000; text-decoration: underline;">57</a>
<a href="#page58" style="color: #000000; text-decoration: underline;">Fehler während des Build</a>	<a href="#page58" style="color: #000000; text-decoration: underline;">58</a>
<a href="#page58" style="color: #000000; text-decoration: underline;">Spalte ist von Typ <code>long</code> anstatt <code>double</code></a>	<a href="#page58" style="color: #000000; text-decoration: underline;">58</a>
<a href="#page59" style="color: #000000; text-decoration: underline;">LZ4/jnizavacpp.dll kann nicht gefunden werden</a>	<a href="#page59" style="color: #000000; text-decoration: underline;">59</a>
<a href="#page59" style="color: #000000; text-decoration: underline;">. . . . .</a>	<a href="#page59" style="color: #000000; text-decoration: underline;">59</a>
<a href="#page59" style="color: #000000; text-decoration: underline;">SSL-Fehler bei der Ausführung</a>	<a href="#page59" style="color: #000000; text-decoration: underline;">59</a>

[. . . . .](#page60)[<a href="#page60" style="color: #000000; text-decoration: underline;">>](#page60)

[Installation Fehlerbehebung](#page63)[<a href="#page63" style="color: #000000; text-decoration: underline;">>Installation Fehlerbehebung](#page63)

[Offline-Installation](#page63)[<a href="#page63" style="color: #000000; text-decoration: underline;">>Offline-Installation](#page63)

[Benutzerdefinierte conda Um](#page64)[<a href="#page64" style="color: #000000; text-decoration: underline;">>Benutzerdefinierte conda Um](#page64)

[Probleme . . . . .](#page64)[<a href="#page64" style="color: #000000; text-decoration: underline;">>Probleme . . . . .](#page64)

[Proxy Issues. . .](#page64)[<a href="#page64" style="color: #000000; text-decoration: underline;">>Proxy Issues. . .](#page64)

# Einleitung

Wie in der [Leitfaden für Erweiterungen und Integrationen](#), KNIME Analytics Platform kann erweitert mit zusätzlichen Funktionen, die durch eine Vielzahl von Erweiterungen bereitgestellt werden und Integrationen. Oft fügt die Installation einer Erweiterung eine Sammlung neuer Knoten zum Knoten hinzu Projektarchiv der KNIME Analytics Platform.

Mit [v4.6 Release](#) von KNIME Analytics Platform stellen wir die Möglichkeit vor, zu schreiben KNIME Knotenerweiterungen komplett in Python. Dies beinhaltet die Fähigkeit, Knoten zu definieren Konfiguration und Ausführung sowie Dialogdefinition. Eine Pythonic API, um diese zu entwerfen nodes ist jetzt verfügbar, sowie Debugging-Funktionalität innerhalb der KNIME Analytics Platform. Dies bedeutet die Bereitstellung von rein-Python KNIME-Erweiterungen mit Knoten – einschließlich ihrer Python-Umgebung, die für die Ausführung benötigt wird – mit einer lokal gebauten Update-Site ist jetzt möglich.

In diesem Leitfaden bieten wir ein Tutorial, um Sie mit dem Schreiben Ihrer KNIME-Knoten mit Python, sowie wie man eine austauschbare Python-Erweiterung mit Ihren Knoten, zusammen mit einer vollständigen Definition der API.

# Quickstart Tutorial

Dieser Abschnitt bietet eine Erweiterungsvorlage und führt Sie durch das Wesentliche Entwicklungsschritte, um Ihnen zu helfen, mit der API gestartet zu werden.

## Voraussetzungen

ANHANG Einrichten **Pixi** .

Pixi ist ein Paket-Management-Tool für Entwickler, die Sie installieren und verwalten Sie die Python Pakete (conda und pypi) für Ihre Erweiterung erforderlich. Es erlaubt Sie die Umgebungen sauber und getrennt für jede Erweiterung zu halten und macht es einfach, bündeln und verteilen Sie Ihre Erweiterung mit all ihren Abhängigkeiten. Beachten Sie, dass die Verwendung **Pixi** ist nicht unbedingt erforderlich, aber es ist sehr empfehlenswert. Verwendung **Conda** noch möglich ist und die Anweisungen finden Sie in einer älteren Version dieser Seite (z. [5.4](#) )

Um mit der Entwicklung von Python Knotenerweiterungen zu beginnen, müssen Sie **Pixi** installiert. Gehen Sie zum [Pixi](#) Website und folgen Sie den Anweisungen dort. Beispielinstallation Befehle:

- ☐ Für Linux und macOS führen Sie in einem Terminal aus: `Curl -fsSL https://pixi.sh/install.sh |`
- ☐ Für Windows, in Power ausführen Shell: `Powerhell -ExecutionPolicy ByPass -c "irm -useb https://pixi.sh/install.ps1 | iex"`
- ☐ Starten Sie Ihre Shell oder Terminal, um sicherzustellen, dass **Pixi** Befehl ist verfügbar.

2. Mit **Pixi** einrichten, herunterladen **knime-python-extension-templat** Repository entweder als eine [Releases](#) [chüss](#) oder es als GitHub-Template verwenden, um Ihr eigenes neues GitHub-Repository zu erstellen. Klicken Sie auf **Verwenden Sie diese Vorlage** Knopf auf der rechten oberen Ecke der [Repository](#) Seite und klonieren Sie Ihr neues Projektarchiv, um es lokal zu bearbeiten.

In der **knime-python-extension-templat** Ordner, Sie sollten die folgende Datei sehen  
Struktur:

Icons

| | â â âTMa icon.png

ÃCE â â âTMa src


$\Psi \perp -$  Erweiterung. Hefe

ÃCE â â âTMa Demos

$\Psi \neq -$  Beispiel\_mit\_Python\_node.knwf

ÃCE â â âTMa knime.yml

- Pixi.toml

ÃCE â â  config.yml

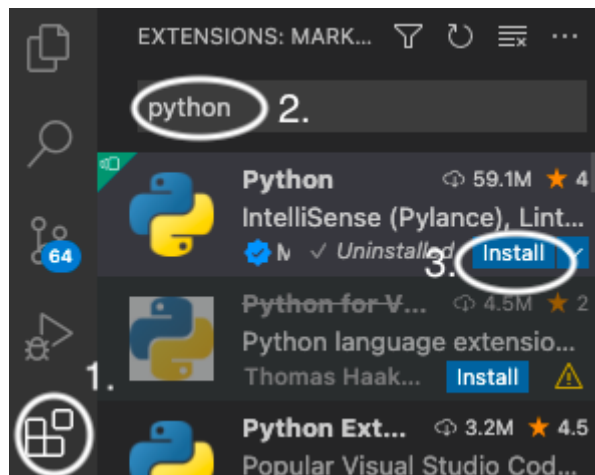
Banananananaranaranananavanananakanavanakanavaw  
nanavanakanavanakanavanakanavanakanavanakanava

$\perp -$  README.md

3. Während der Entwicklung können Sie die Quelldateien in jedem Texteditor bearbeiten. Um die Verwendung von Autocompletion für die API, sowie zu ermöglichen Debugging über die Paket, empfehlen wir, einen Editor zu verwenden, der Umgebungen als Python einstellen kann Dolmetscher. Hier sind die Setupschritte für **Visual Studio Code** :

## Debugpy

- ☐ [Downloads](#) und Visual Studio Code installieren
- ☐ Installieren Sie die Python Erweiterung



- ☐ In der unteren rechten Ecke des Editors wählen Sie den Python-Interpreter, den Sie wollen bei der Entwicklung zu verwenden. Nach Schritt 4 von Tutorial 1 haben Sie die Umwelt verfügbar. Wenn Sie mit der Standardumgebung starten möchten, können Sie Ausführung:

## Pixi installieren

die die bereitgestellte Standardumgebung installiert. Sie können noch mehr hinzufügen  
Pakete später. Um die Umgebung auszuwählen, klicken Sie auf die Python-Version in der  
unten rechts Ecke des Editors. Ein Menü erscheint mit dem verfügbaren Python  
Dolmetscher. In der Einstellung "Select Python Interpreter" wählen Sie die Umgebung

'default':Pixi . Wenn es nicht automatisch angezeigt wird, können Sie es auch manuell hinzufügen indem Sie auf "Interpreterpfad eingeben" und die Auswahl der Python ausführbar in der .pixi/envs/default Verzeichnis. Damit können Sie den Linter verwenden und autocomplete Funktionen Ihrer IDE. Durch die Auswahl der Umgebung können Sie zur vollen Nutzung der Autokompletion.



## Schreiben Sie Ihren ersten Python-Knoten von Grund auf

Dies ist ein Schnellstart-Guide, der Sie durch die wesentlichen Schritte des Schreibens und Laufens gehen wird

Ihre erste Python Knotenerweiterung mit einem einzigen Knoten. Wir werden knime-python-Verlängerungstemplat als Grundlage. Die Schritte des Tutorials, die eine Änderung der Python-Code in Erweiterung. Hefe haben entsprechende Kommentare in der Datei, zum Komfort.

Für einen umfangreichen Überblick über die vollständige API finden Sie unter [Python: Volle API](#) [Lesen Sie die Docs Seite](#).  
[Abschnitt, sowie unsere](#)

ANHANG KNIME installieren Analytics Platform Version 5.5.0 oder höher.

2. Gehen Sie Datei → KNIME installieren Erweiterungen... , Geben Sie "Python" im Suchfeld ein und suchen Sie nach KNIME Python Extension Development (Labs) . Alternativ können Sie manuell navigieren in der KNIME Laborerweiterungen kategorie und finden Sie die Erweiterung dort. Wählen Sie es und mit der Installation fortfahren.

3. Die knime-python-extension-templat wird Ihre neue Erweiterung sein. Vertraue dich mit den in diesem Ordner enthaltenen Dateien, insbesondere:

- ☐ knime.yml , die wichtige Metadaten über Ihre Erweiterung enthält.
- ☐ Erweiterung. Hefe , die Python Definitionen der Knoten Ihrer Erweiterung enthält.
- ☐ config.yml , nur außerhalb des Ordners, der die Informationen enthält, die bindet Ihre Erweiterung und die entsprechende Conda /Python-Umgebung mit KNIME Analyseplattform.

L 347 vom 20.12.2013, S. 1). Erstellen Sie eine Python-Umgebung für Ihre Erweiterung mit [knime-python-Basis metapackage](#), zusammen mit der [Knoten-Entwicklung API knime-extension](#) für das KNIME Analytics Platform, die Sie verwenden. Sie können dies einfach tun, indem Sie die folgenden Pixi Befehl in Ihrem Terminal (auch in einem Terminalfenster im VS-Code):

Pixi installieren

Das funktioniert, weil die Pixi.toml Datei in der knime-python-extension-templat

Ordner enthält die erforderlichen Pakete für die neueste KNIME Analytics Platform Version  
in der Standard-Umwelt.

Um zusätzliche Pakete zu installieren, empfehlen wir für Ihren speziellen Anwendungsfall die Verwendung der  
conda-forge-Kanal, der standardmäßig hinzugefügt wird.

```
Pixi add
```

5. Bearbeiten Sie die `config.yml` Datei im Wurzelordner der `knime-python-extension-`  
Vorlage (für dieses Beispiel existiert die Datei bereits mit vorgefüllten Feldern und Werten, aber  
Sie müssten sie anpassen). Der Inhalt sollte wie folgt sein:

```
src: /src
conda_env_path: /.pixi/envs/default
debug_mode: true
```

Wo:

- ☐ `knime.yml` sollte ersetzt werden durch `Gruppe` und `Name` Werte, die in `knime.yml`, kombiniert mit einem Punkt.

Für unsere Beispielerweiterung, den Wert für `Gruppe` ist `Org.tutorial`, und der Wert für `Name` ist `erste_extension`, daher die Platzhalter sollte ersetzt durch `org.tutorial.first_extension`.

- ☐ Die `Src` Feld sollte den Pfad zum Ordner angeben, in dem `knime.yml` von dir Python Erweiterung befindet sich. Dies ist in der Regel der Hauptordner Ihrer Erweiterung.

Zum Beispiel `/Benutzer/Bobby/Development/python_extensions/knime-python-Verlängerungstemplat/Src`

- ☐ In ähnlicher Weise Das ist der Fall. `src` Feld sollte den Pfad zur Umgebung angeben in der `Pixi.toml` Datei, erstellt in Schritt 4. Wenn Sie die Standardumgebung verwenden, die mit dem `knime-python-extension-templat`, der Weg sollte `/Benutzer/Bobby/Development/python_extensions/knime-python-extension-Vorlage/.pixi/envs/default`. Dies kann jedoch jeder Weg zu einem gültigen Python /Conda-Umwelt.

Weitere Informationen zum `Pixi.toml` Datei und `Pixi` sich in der [Pixi Dokumentation](#), die detaillierte Anleitung zur Konfiguration von Umgebungen bietet und die Verwaltung von Abhängigkeiten.

Die `Debug_mode` ist ein optionales Feld im `config.yml` Datei. Wenn eingestellt `wahr`, instructs KNIME Analytics-Plattform, um die neuesten Änderungen in der Konfiguration und Ausführung Methoden Ihrer Python-Knotenklasse, wenn diese Methoden aufgerufen werden. Das ist nützlich während der Entwicklung, kann aber Auswirkungen Knoten Reaktionsfähigkeit.

Absetzen `Debug_mode` wird die Reaktionsfähigkeit Ihrer Knoten beeinflussen.

6. Wir müssen die KNIME Analytics Plattform wissen lassen, wo die `config.yml` ist, um ermöglichen es, unsere Erweiterung und ihre Python-Umgebung zu nutzen. Um dies zu tun, müssen Sie bearbeiten die `knime.ini` Ihrer KNIME Analytics Plattform Installation, die sich an `to-your-KAP > /knime.ini`.

Am Ende des `knime.ini` Datei, ersetzen mit dem eigentlichen Weg zu Ihrem `config.yml` Datei:

```
-Dknime.python.extension.config =
```

Benutzen Sie Vorwärtsspritzen ( `/` ) für alle Betriebssysteme, einschließlich Windows. Die `knime.ini` Datei befindet sich im Installationsordner von KNIME Analytics Plattform.

7. Starten Sie Ihre KNIME Analytics Plattform.

8. Die "My Template Node" sollte nun im Knoten-Repository sichtbar sein.

ANHANG Importieren und öffnen `Beispiel_mit_Python_node.knwf` Arbeitsablauf aus dem `Demonstrieren` Ordner, der einen Testknoten enthält:

a. Vertraue dich mit der Tabellenstruktur und Daten.

B. Studieren Sie den Code in `Erweiterung.Hefe` und vergleichen Sie es mit dem Knoten, den Sie in KNIME sehen Analyseplattform. Insbesondere verstehen Sie, wo der Knotenname, Beschreibung, Ein- und Ausgänge werden im Code definiert.

c. Führen Sie den Knoten aus und überprüfen Sie, ob er eine Ausgabetabelle erstellt.

10. Erstellen Sie Ihren ersten Konfigurationsdialog:

In `Erweiterung.Hefe`, die Definitionen der Parameter nicht einkommen (gekennzeichnet durch das "Tutorial Schritt 10" Kommentar). Starten Sie Ihre KNIME Analytics Plattform, um Ihren Knoten aus dem node repository in the workflow, and doubleclick the node to see the konfigurable Parameter.

Nehmen Sie eine Minute, um die Namen, Beschreibungen und Standardwerte in `Erweiterung.Hefe` mit dem, was im Knoten-Dialog angezeigt wird.

#### 11. Fügen Sie Ihren ersten Port hinzu:

Um dem Knoten eine zweite Eingabetabelle hinzuzufügen, folgen Sie diesen Schritten (markiert durch das "Tutorial Schritt 11" Kommentar:

- a. Unzufrieden `@knext.input_table` Dekorateure in SpanierErweiterung. Hefe .
  - B. Aktualisieren der Konfiguration Verfahren zur Bearbeitung des Schemas der zweiten Eingabetabelle.
  - c. Ändern der Ausführung Verfahren zur Verarbeitung von Daten aus beiden Eingabetabellen.
- KNIME Analytics Platform zur Anwendung der Änderungen.

#### 12. Fügen Sie dem Knoten einige Funktionalität hinzu:

Mit den folgenden Schritten werden wir eine neue Spalte an die erste Tabelle anhängen und die neue Tabelle (die Linien, die geändert werden müssen, sind durch den "Tutorial Step 12' gekennzeichnet ' Kommentar:

- a. Um nachgeschaltete Knoten des geänderten Schemas zu informieren, müssen wir es im Rückmeldung der Konfiguration Methode; dazu fügen wir Metadaten über eine Spalte zum Ausgabeschema.
- B. Alles andere ist in der Ausführung Methode:
  - wir transformieren beide Eingabetabellen in Pandas-Datenrahmen und fügen eine neue Spalte zum ersten Datenrahmen
  - wir wandeln diesen Datenrahmen wieder in eine KNIME-Tabelle um und geben ihn zurück

#### 13. Verwenden Sie Ihre Parameter:

- a. In der Ausführung Methode, die durch den "Tutorial Step 13' markierten Linien nicht zu erreichen ' Kommentar.
- B. Verwenden Sie einen Parameter, um einige Tabelleninhalte zu ändern; wir verwenden eine Lambda-Funktion für eine reihenweise Multiplikation mit Doppel-Parameter Parameter.

#### 14. Starten Sie Protokollierung und Einstellung Warnungen:

Die von "Tutorial Step 14" markierten Linien werden in der Ausführung Methode:

- a. Verwenden Sie die `LOGGER` Funktionalität zum Protokollieren von Nachrichten für Debugging oder Benutzerinformationen.
- B. Verwenden Sie die `ausführen_context.set_warning("A warnung")` um die Benutzer über ungewöhnliches Verhalten.
- c. Wenn Sie wollen, dass der Knoten scheitert, erhöhen Sie eine Ausnahme. Zum Beispiel:

```
ValueError("Dieser Knoten hat aufgrund eines Fehlers versagt.")
```

ANHANG Herzlichen Glückwunsch, Sie haben Ihren ersten funktionierenden Knoten komplett in Python gebaut! Du bist jetzt bereit, erweiterte Funktionen zu erkunden und die Fähigkeiten Ihrer Erweiterung zu erweitern.

# Python Node Erweiterung Setup

Eine Python Knoten-Erweiterung muss eine YAML-Datei namens `knime.yml` enthalten, die allgemeine Informationen über die Knotenerweiterung, die Python-Modul zu laden, und welche conda Umwelt sollte mit der Erweiterung gebündelt werden.

`knime.yml` :

```
Name: myextension # Wird mit der group_id verkettet, um die Extension ID zu bilden
Autor: Jane Doe
Pixi_toml_path: # Path to the Pixi Environment toml, from which
die Umgebung für diese Erweiterung wird bei der Bündelung gebaut
extension_module: extension # Das .py Python-Modul mit den Knoten Ihres
Verlängerung
Beschreibung: My New Extension # Menschenlesbarer Bündelname / Beschreibung
Long_description: Diese Erweiterung bietet Funktionalität, die jeder haben möchte. #
Text, der die Erweiterung beschreibt (optional)
group_id: org.knime.python3.nodes # Will be concatenated with the name to form the
Kennnummer
Version: 0.1.0 # Version dieser Python Knotenerweiterung. Muss Dreikomponenten verwenden
semantische Versionierung zum Einsatz.
Anbieter: KNIME AG, Zürich, Schweiz # Who bietet die Erweiterung an
License_file: LICENSE.TXT # Best Practice: setzen Sie Ihre LICENSE. TXT neben dem Strick. yml;
andernfalls müssen Sie auf path/to/LICENSE.txt wechseln
#Optional: Wenn Sie keine Abhängigkeiten von anderen Erweiterungen haben, brauchen Sie nicht
feature_dependencies und ihre Einträge
feature_dependencies:
- org.knime.features.chem.types 5.7.0 # Wenn Sie die Version angeben möchten - beachten Sie, dass
die Version ist größer 5.7.0
- org.knime.features.chem.types # Wenn die Version nicht wichtig ist
```

Die `id` der Erweiterung wird von der Form `Gruppe_id.name`. Es muss eine eindeutige Kennung sein für Ihre Erweiterung, so ist es eine gute Idee, Ihren Benutzernamen oder Firmen-URL zu kodieren gefolgt durch eine logische Struktur als `Gruppe` um zu verhindern `id` Blödsinn. Zum Beispiel ein Entwickler von KNIME kann seine URL kodieren `Org.knime` und hinzufügen `Python`, die Erweiterung ist Mitglied `Knoten`, die Teil von `Python`.

Funktionsabhängigkeiten: Wenn Ihre Erweiterung von einer anderen Erweiterung abhängt, können Sie sie als einen Kugelpunkt `feature_dependancies`. Optional können Sie eine bestimmte Mindestversion hinzufügen zu ihm.

Beispiel: Sie verwenden Datentypen wie `SmilesValue` von `KNIME Basischemie-Typen & Nodes` Erweiterung in Ihrer Erweiterung. Sie haben diese Erweiterung bereits installiert und möchten sicherstellen, dass alle, die Ihre Erweiterung verwenden, auch diese Erweiterung installiert haben. Dann kannst du gehen, [Hilfe Über KNIME Analytics Plattform > Installationsdetails](#) und überprüfen Sie die `id` von `KNIME`

Basischemie-Typen & Nodes , die org.knime.features.chem.types.feature.group .

Nehmen Sie die id ohne .feature.group und Sie haben den String der Funktion Abhängigkeit: org.knime.features.chem.types

Anmerkung: Pixi\_toml\_path Feld, das den Pfad zum Pixi.toml Datei

Konfiguration der Umgebung, die durch Ihre Erweiterung benötigt wird, wird benötigt, wenn Sie Ihre Bündelung Erweiterung. Bei der Entwicklung nutzt die KNIME Analytics Platform die in die config.yml Datei.

Der Weg, der die knime.yml wird dann auf die Pythonpath , und die Erweiterung

Das in der YAML spezifizierte Modul wird von der KNIME Analytics Platform importiert. Einfuhr

. Dieses Python-Modul sollte die Definitionen von KNIME-Knoten enthalten.

Jede Klasse mit @knext.node innerhalb dieser Datei wird verfügbar in KNIME Analytics Platform als dedizierter Knoten.

Empfohlene Projektordnerstruktur:

```
.
Icons
| | â â âTMa icon.png
Ãœ â â âTMa src
Ψ ↳ Erweiterung. Hefe
Ãœ â â âTMa Demos
Ψ ψ ≈ — Beispiel_mit_Python_node.knwf
Ãœ â â âTMa knime.yml
- Pixi.toml
Ãœ â â â config.yml
~~~~~
↳ — README.md
```

[Tutorial 1](#page6)

Vgl. oben zum Beispiel.

Entwicklung und Vertrieb

Wenn Sie Ihre Python-Erweiterung entwickeln, können Sie sie lokal ausführen und debuggen, indem Sie die knime.python.extension.config Systemeigenschaft in der KNIME Analytics Platform

knime.ini zu zeigen, config.yml , oder in den VM-Argumenten der Startkonfiguration in

Eclipse. Siehe [Python ausführbar](#page53) und [Python ausführbar](#page52)

Abschnitte am Ende dieser Führung für weitere Informationen.

Um Ihre Extension Python mit anderen zu teilen, lesen Sie bitte die [Erweiterungsnummern](#page48)

Abschnitt.



- ☐ `knext.NodeTyp.MANIPULATOR` : ein Knoten, der Daten manipuliert.
- ☐ `knext.NodeTyp.LEARNER` : ein Knoten, der ein Modell lernt, das typischerweise von einem `PREDICTOR`.
- ☐ `knext.NodeTyp.PREDICTOR` : ein Knoten, der mit einem Modell eines `LEARNER`.
- ☐ `knext.NodeTyp.SOURCE` : ein Knoten, der Daten erzeugt.
- ☐ `knext.NodeTyp.SINK` : ein Knoten, der Daten verbraucht.
- ☐ `knext.NodeTyp.VISUALIZER` : ein Knoten, der Daten visualisiert.
- ☐ `knext.NodeType.OTHER` : ein Knoten, der keinen der anderen Knotentypen passt.

• **Icon\_path** : Modul-relativer Pfad zu einer 16x16 Pixel PNG-Datei, um als Symbol zu verwenden.

• **Kategorie** : definiert den Pfad zum Knoten innerhalb der KNIME Analytics Platform's `Node Repository` .

## Benutzerdefinierte Port-Objekte definieren

Neben Tabellen kann ein Knoten auch andere Port-Objekte konsumieren oder produzieren und es ist möglich, definieren Sie benutzerdefinierte Port-Objekte für Ihre Erweiterung. Sie können dies durch Verlängerung `knext.PortObject` und `knext.PortObjectSpec` mit Ihrer benutzerdefinierten Implementierung. Um diese Objekte in Ihr Knoten, Sie müssen einen benutzerdefinierten Porttyp über den definieren `knext.port_type` Funktion, die Ihr `PortObject` und `PortObjectSpec` Klassen sowie ein menschenlesbarer Name für Ihren Porttyp und optionale ID. Hier ist ein Beispiel:

Beginnen wir mit dem `PortObjectSpec` :

```
import knime. Verlängerung als knext

Klasse MyPortObjectSpec (knext.PortObjectSpec):
def __init__(self, spec_data: str) -> Keine
Super().__init__()
selbst._spec_data = spec_data

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

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

@Prop.
def spec_data(self) -> str:
zurück._data
```

Die `spec_data` serienmäßig und `spec_data` Das ist das `spec_data` Methoden werden vom Rahmen verwendet, um die spez.

Anmerkung: Die `spec_data` Das ist das `spec_data` Verfahren muss ein `spec_data` Klassenmethod `spec_data`.

Die `spec_data` Daten `spec_data` Eigenschaft ist nur ein Beispiel für benutzerdefinierten Code und Sie können beliebig hinzufügen Methoden zu Ihrer Spezifikation Klasse, wie Sie sehen fit.

Als nächstes implementieren wir die `PortObject` :

```
Import Pickle

Klasse MyPortObject(knext. PortObject:
def __init__(self, spec: MyPortObjectSpec, model) -> Keine
super().__init__(spec)
selbst._model = Modell

def serialize(self) -> Bytes:
zurück pickle.dumps(self._model)

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

def predict(self, data):
zurück selbst._model.predict(data)
```

Die `PortObject`-Klasse muss eine `spec_data` serienmäßig und `spec_data` Das ist das `spec_data` Verfahren, die von der Rahmen, um das Objekt zu bestehen und wiederherzustellen. Beachten Sie erneut, dass `spec_data` Das ist das `spec_data` muss ein

Klassenmethod `__init__`.

Die `__str__` Vorhersage Eigenschaft ist wieder nur ein Beispiel für benutzerdefinierten Code, dass Ihr Port-Objektklasse kann enthalten.

Schließlich erstellen wir einen benutzerdefinierten Porttyp, der als Eingabe oder Ausgabe eines Knotens verwendet wird:

```
my_model_port_type = knext.port_type(name="Mein Modell Hafentyp",
object_class=MyPortObject, spec_class=MyPortObjectSpec)
```

Die `knext.port_type` Methode bindet die `PortObject` und `PortObjectSpec` zusammen und bietet einen human lesbaren Namen, der sich auf den benutzerdefinierten Porttyp bezieht.

Es ist auch möglich, eine benutzerdefinierte ID für den Porttyp über die `id` Argument. Anmerkung: ID muss einzigartig sein. Wenn Sie keine benutzerdefinierte ID bereitstellen, erzeugt das Framework eine der Format `Ihr_extension_id.your_module_name.your_port_object_class_name`. Zum Beispiel, wenn Ihre Erweiterung hat den Ausweis `org.company.extension` und Sie implementieren eine `MyPortObject` in der Modul `Verlängerung`, dann wird die generierte ID `org.company.extension.extension.MyPortObject`.

Beachten Sie, dass es auch Verbindungsport-Objekte, die nicht-serialisierbare Objekte halten können. Du

[Informationen dazu finden](#) [API-Dokumentation](#) für `knime.extension.ConnectionPort` Gegenstand.

Überprüfen Sie den nächsten Abschnitt, um zu lernen, wie Sie Ihren benutzerdefinierten Porttyp als Eingabe oder Ausgabe erklären Ihr Knoten.

## Keine Port-Konfiguration

Die Eingabe- und Ausgabeanschlüsse eines Knotens können durch Dekorieren der Knotenklasse mit

`@knext.input_table`, `@knext.input_port`, und `@knext.output_table` und `@knext.output_port`. Mit der `@knext.output_image` dekorator. Zusätzlich sollte ein Knoten, der einen Blick erzeugt, mit dem `@knext.output_view` dekorator.

Diese Port-Dekoratoren haben folgende Eigenschaften:

- sie nehmen `Name` und `Beschreibung` Argumente, die im Knoten angezeigt werden Beschreibungsbereich innerhalb der KNIME Analytics Platform;
- sie müssen positioniert werden nach die `@knext.node` Dekorator und vor die dekorierten Objekt (z.B. die Knotenklasse);

- ihre Bestellung bestimmt die Reihenfolge der Port-Steckverbinder des Knotens in KNIME Analytics Plattform.

Die `@knext.input_table` und `@knext.output_table` Dekorateure konfigurieren den Hafen konsumieren und produzieren jeweils einen KNIME-Tisch. Die `@knext.output_image` Dekorateur konfiguriert den Port zur Erstellung eines PNG- oder SVG-Bildes.

Wenn Sie andere Daten empfangen oder senden möchten, z.B. ein geschultes maschinelles Lernmodell, verwenden `@knext.input_port` und `@knext.output_port`. Diese Dekorateure ein zusätzliches Argument `port_type`, verwendet, um die Art von Port-Objekten entlang dieser Port-Verbindung zu identifizieren. Nur Häfen mit gleichem `port_type` kann angeschlossen werden. Sehen Sie den vorherigen Abschnitt, um zu lernen, wie geben Sie Ihren eigenen Porttyp an.

Die Portkonfiguration bestimmt die erwartete Signatur der Konfiguration und Ausführung Methoden:

- In der Konfiguration Methode, das erste Argument ist ein `ConfigurationContext`, gefolgt von Argument pro Eingangsport. Die Methode wird voraussichtlich zurück **so viele Parameter wie es verfügt über Ausgangsports**. Die Argument- und Rückgabewerttypen entsprechen der Eingangs- und Ausgangsanschlüsse sind:

- ☐ für **TabellePorts**, der Argument/Return-Wert muss vom Typ sein `knext.Schema`. Wenn Rückgabetable besteht aus nur einer Spalte, der Rückgabewert kann auch vom Typ `knext.Spalte`;
- ☐ für **Bild** Ports, der Argument/Return-Wert muss vom Typ sein `knext.ImagePortObjectSpec` mit dem entsprechenden Bildformat konfiguriert
- ☐ für **benutzerdefinierte** Ports, der Argument/Return-Wert muss von Ihrem benutzerdefinierten Durchführung `knext.PortObjectSpec`. Wenn wir das Beispiel von früherer Abschnitt, der Typ wäre `MyPortObjectSpec`.

Beachten Sie, dass die Reihenfolge der Argumente und Rückgabewerte der Reihenfolge der Argumente entsprechen muss die Eingabe- und Ausgabeport-Deklarationen über die Dekorateure.

- Die Argumente und erwarteten Rückgabewerte der Ausführung Methode folgt der gleichen Schema: ein Argument pro Eingangsport, ein Rückgabewert pro Ausgangsport. Für Bild Der zurückgegebene Wert muss vom Typ sein `Bytes`.

Beispiele wie zu verwenden `knext.Schema` und `knext.Spalte`` (siehe [API](#) :

```
def konfigurieren (selbst, config_context): # keine Eingabetabelle
""" Dieser Knoten erstellt eine Tabelle mit einer einzigen Spalte """
ktype = knext.string()
ODER
ktype = knext.int32() # OR knext.double(), knext.bool_(), knext.list_(knext.string()),
knext.struct(knext.int64(), knext.bool_()),...
ODER
Einfuhrdatum
ktype = Datumszeit.datetime
Zurück knext. Spalte(KTyp, "Date and Time")
```

```
def konfigurieren (selbst, config_context): # keine Eingabetabelle
""" Dieser Knoten erstellt zwei Tabellen mit je zwei Spalten """
ktype1 = knext.string()
import knime.types.chemistry as cet # needs the extension `KNIME Base Chemistry Types
& Nodes` installiert
ktype2 = cet.SdfValue
Schema1 = knext.Schema([ktype1, ktype2], ["Column with Strings", "Column with Sdf"])
schema2 = knext. Schema([ktype1, ktype2], ["Eine andere Spalte mit Strings", "Eine andere Spalte
Spalte mit Sdf"])
zurück Schema1, Schema2
```



Alle unterstützten Arten Ihrer aktuellen Umgebung können durch Drucken erhalten werden

```
knime.api.schema.supported_value_types()          oder
knime.extension.supported_value_types() `        .
```

Hier ist ein Beispiel mit zwei Eingangsanschlüssen und einem Ausgangsport. Siehe die vorherige Sitzung für

Definitionen `MyPortObject` , `MyPortObjectSpec` und `my_model_port_type` .

```
@knext.node("My Predictor", node_type = knext.NodeType.PREDICTOR, icon_path = "icon.png",
Kategorie = "/")
@knext.input_port("Trained Model", "Trained Phantasie Maschine Lernmodell",
port_type=my_model_port_type)
@knext.input_table("Data", "Die Daten, auf die vorhergesagt werden soll")
@knext.output_table("Output", "Resulting table")
Klasse MyPredictor():
def konfigurieren(selbst, config_context: knext. ConfigurationContext, input_spec:
MyPortObjectSpec, table_schema: knext.Schema) -> knext. Schema:
# Wir werden eine Spalte des Typs Doppel zur Tabelle hinzufügen
zurück table_schema.append(knext.Column(knext.double(), "Predictions"))
# Wenn Sie Typen verwenden möchten, die KNIME bekannt sind, aber die keine spezielle KNIME haben
Typ, können Sie verwenden:
# Import datetime
# return table_schema.append(knext.Column(datetime.datetime, "Date and Time"))

def ausführen(self, exec_context: knext. ExecutionContext, geschult_model: MyPortObject,
input_table: knext.Table) -> knext. Tabelle:
Vorhersagen = train_model.predict(input_table.to_pandas())
output_table = input_table
output_table["Predictions"] = Vorhersagen
zurück knext.Table.from_pandas(output_table)
```

Beispiel mit zwei Bildausgängen.

```
@knext.node("My Image Generator", node_type=knext.NodeType.SOURCE, icon_path="icon.png",
Kategorie = "/")
@knext.output_image(name="PNG Ausgabebild", Beschreibung="Ein Beispiel PNG Ausgabebild")
@knext.output_image(name="SVG Ausgabebild", Beschreibung="Ein Beispiel SVG Ausgabebild")
Klasse ImageNode:
def konfigurieren (selbst, config_context):
    zurück (
    knext.ImagePortObjectSpec(knext.ImageFormat.PNG),
    knext.ImagePortObjectSpec(knext.ImageFormat.SVG),
    )

def ausführen(self, exec_context):
    x = [1, 2, 3, 4, 5]
    y = [1, 2, 3, 4, 5]
    fig, ax = plt.subplots(figsize=(5, 5), dpi=100)
    ax.plot(x, y)

    Puffer_png = io.BytesIO()
    plt.savefig(buffer_png, format="png")

    Puffer_svg = io.BytesIO()
    plt.savefig(buffer_svg, format="svg")

    zurück (
    Puffer_png.getvalue(),
    Puffer_svg.getvalue(),
    )
```

Alternativ können Sie die `Input_ports` und `Ausgabe_ports` Attribute Ihres Knotens Klasse (auf Klassen- oder Instanzebene) für eine feinkörnige Kontrolle.

## Angabe der Knotenkategorie

Jeder Knoten in Ihrer Python Knotenerweiterung wird eine Kategorie über die `Kategorie` Parameter von `@knext.node` dekorator, der diktiert, wo sich der Knoten im Knoten befindet Projektarchiv der KNIME Analytics Plattform. Ohne eine explizite Kategorie wird der Knoten platziert in der Wurzel des Knoten-Repository, also sollten Sie **immer** eine Kategorie für jeden Knoten angeben.

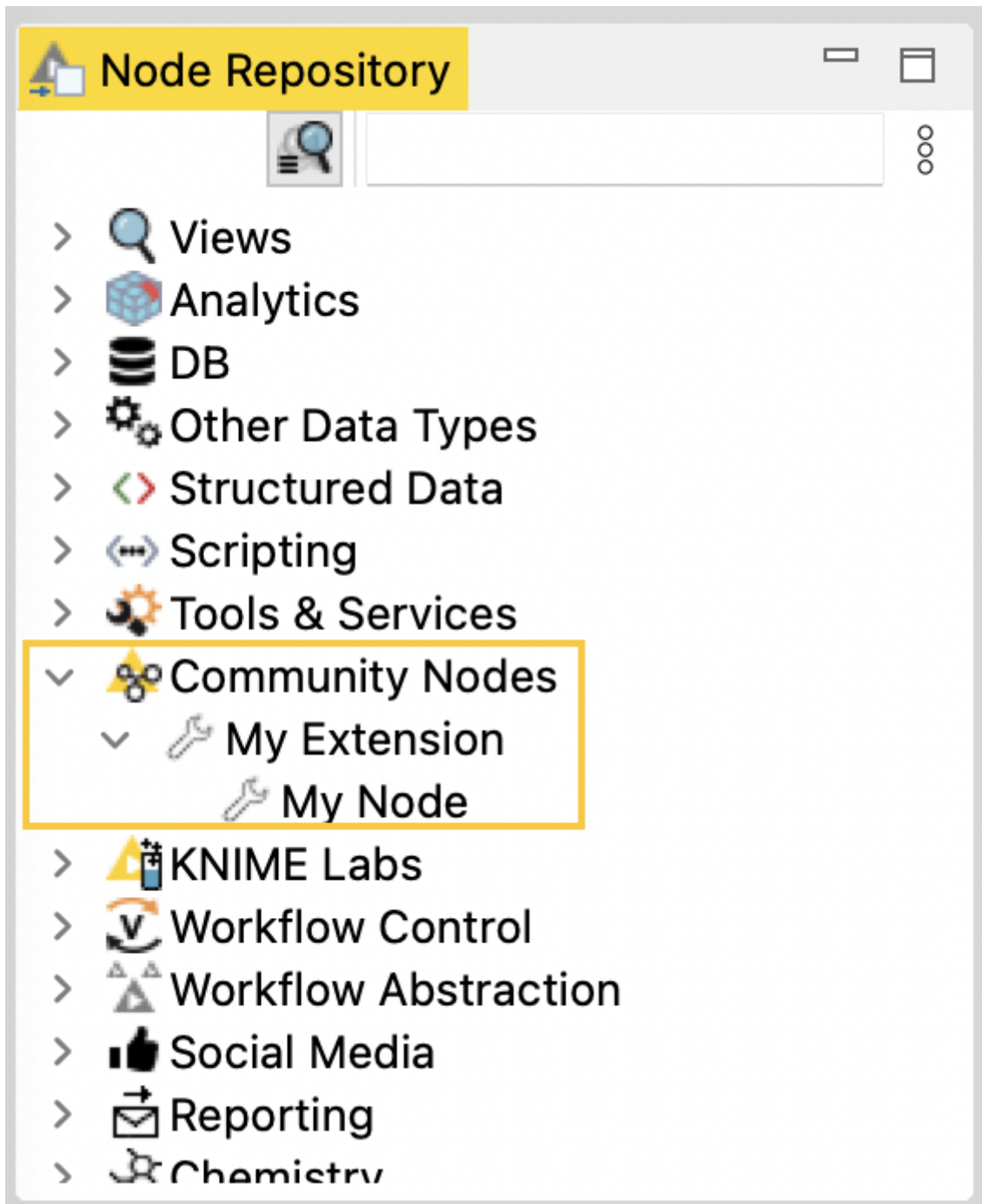
Um eine benutzerdefinierte Kategorie für die Knoten Ihrer Erweiterung zu definieren, können Sie die `knext.kategorie` Hilfsfunktion. Wenn Autocompletion in Ihrer IDE aktiviert ist, sollten Sie in der Lage sein, die Liste der erwarteten Parameter der Funktion zusammen mit ihren detaillierten Beschreibung.

Wenn Sie ein `Community Entwickler`, Sie sollten die **Gemeinschaftsvorschriften** Kategorie als Stammkategorie Ihrer Python Knotenerweiterungen. Dies geschieht durch Angabe der `path="/community"` Parameter der `knext.kategorie` Funktion:

```
import knime. Verlängerung als knext

my_category = knext.category(
    path="/community",
    level_id="extension",
    Name="Meine Erweiterung",
    Beschreibung="My Python Node Extension.",
    Icon="icon.png",
)

@knext.node(
    Name="My Node",
    node_type=knext.NodeType.PREDICTOR,
    Icon_path="icon.png",
    Kategorie = my_catego
)
...
Klasse MyNode():
...
.
```



Beachten Sie, dass es möglich ist, Ihre benutzerdefinierte Kategorie in Unterkategorien weiter zu teilen. Dies ist nützlich

wenn zum Beispiel Knoten Ihrer Erweiterung basierend auf ihrer Funktionalität gruppiert werden können. Zuerst

eine Elternkategorie für die Knotenerweiterung definieren, können Sie sie dann als die

Pfad Parameter

bei der Definition der Unterkategorien:

```
import knime. Verlängerung als knext

# die Kategorie und ihre Unterkategorien definieren
main_category = knext.category(
    path = "/community",
    level_id = "extension",
    name = "scikit-learn Extension",
```

```
Beschreibung="Nodes implementieren verschiedene scikit-learn Algorithmen.",
```

```
Icon="icon.png",
```

```
)
```

```
beaufsichtigt_kategorie = knext.kategorie(
```

```
Pfad = main_category,
```

```
level_id="supervised_learning",
```

```
name="Supervised Learning",
```

```
Beschreibung="Nodes for betreut learning.",
```

```
Icon="icon.png",
```

```
)
```

```
nichtupervised_category = knext.category(
```

```
Pfad = main_category,
```

```
level_id="unsupervised_learning",
```

```
name="Unsupervised Learning",
```

```
Beschreibung="Nodes for unupervised learning.",
```

```
Icon="icon.png",
```

```
)
```

```
# Knoten der Erweiterung definieren
```

```
@knext.node(
```

```
name="Logistic Regression Learner",
```

```
node_type=knext.NodeType.SINK
```

```
Icon_path="icon.png",
```

```
Kategorie(n)
```

```
)
```

```
...
```

```
Klasse LogisticRegressionLearner():
```

```
...
```

```
@knext.node(
```

```
Name="SVM Lerner",
```

```
node_type=knext.NodeType.SINK
```

```
Icon_path="icon.png",
```

```
Kategorie(n)
```

```
)
```

```
...
```

```
Klasse SVMLearner():
```

```
...
```

```
@knext.node(
```

```
Name="K-Means Learner",
```

```
node_type=knext.NodeType.SINK
```

```
Icon_path="icon.png",
```

```
Kategorie=unsupervised_kategorie
```

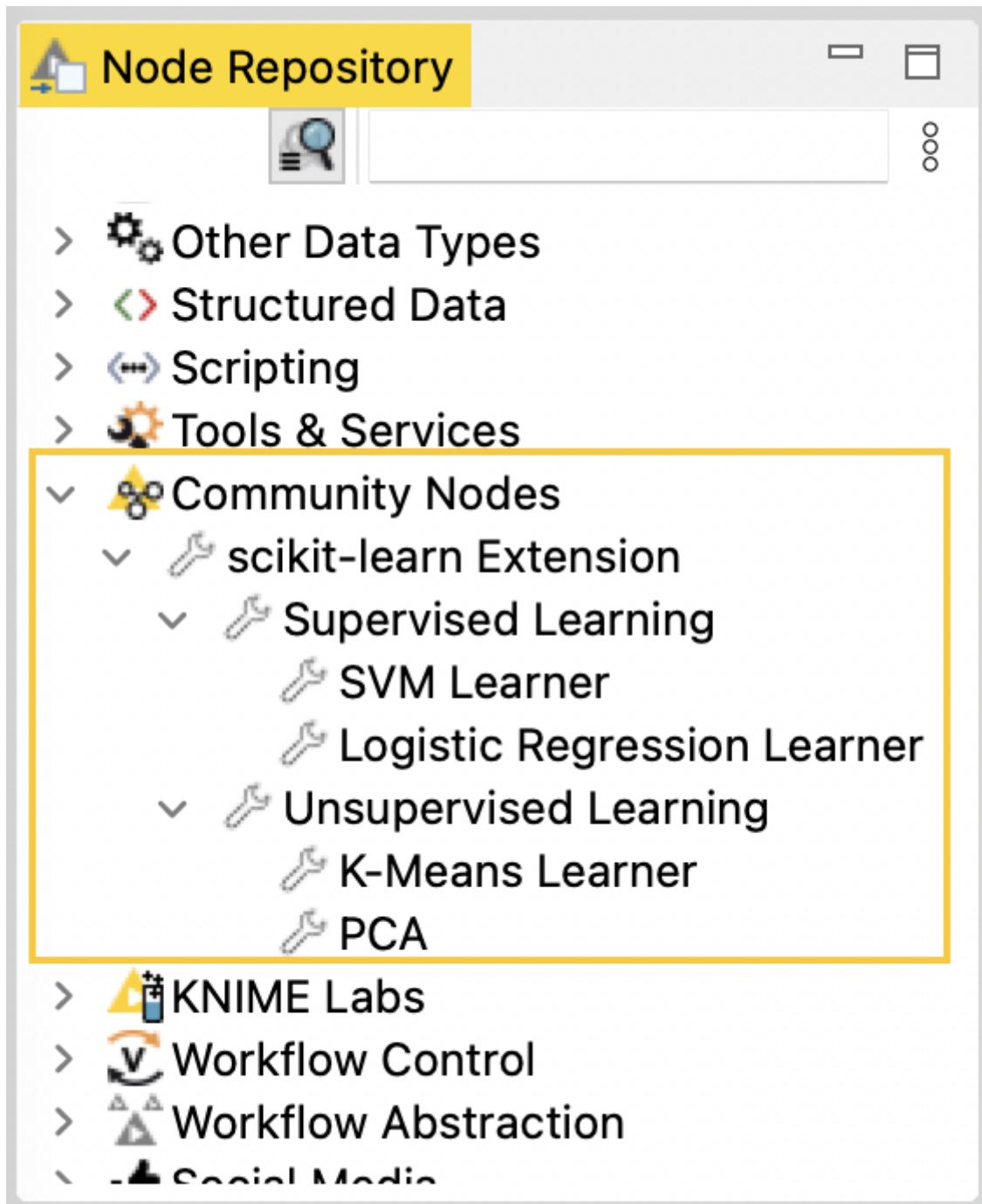
```
)
```

```
...
```

```
Klasse KMeansLearner():
```

```
...
```

```
@knext.node(  
    Name="PCA Learner",  
    node_type=knext.NodeType.SINK  
    Icon_path="icon.png",  
    Kategorie=unsupervised_kategorie  
)  
...  
Klasse PCALearner():  
...  
.
```



## Definieren des Konfigurationsdialogs des Knotens



Um der Kürze willen, in den folgenden Code-Snippets verzichten wir repetitive Teile des Codes, dessen Dienstprogramm bereits festgelegt wurde und vorhin demonstriert.

Um der Funktionalität Ihres Knotens eine Parametrierung hinzuzufügen, können wir seine Konfigurationsdialog. Die dort angezeigten benutzerkonfigurierbaren Parameter und deren Werte innerhalb der `Ausführung Verfahren des Knotens über selbst.param_name`, die folgenden Parameterklassen, die in `andere` :

- `knext.IntParameter` für ganze Zahlen:

☐ Unterschrift:

```
knext.IntParameter(
    Label=None,
    Beschreibung=Nein,
    default_value=0,
    min_value=None,
    max_value=None,
    seit_version=Nein,
)
```

☐ Definition innerhalb einer Node/Parameter-Gruppenklasse:

```
no_steps = knext.IntParameter("Anzahl der Schritte", "Die Anzahl der Wiederholungen
Schritten.", 10, max_value=50)
```

☐ Nutzung innerhalb der `Ausführung Verfahren der Knotenklasse:`

```
für i in range(self.no_steps):
    ↳ do something
```

- `knext.DoubleParameter` für schwimmende Punktzahlen:

☐ Unterschrift:

```
knext.DoubleParameter(
    Label=None,
    Beschreibung=Nein,
    default_value=0.0,
    min_value=None,
    max_value=None,
    seit_version=Nein,
)
```

- ☐ Definition innerhalb einer Node/Parameter-Gruppenklasse:

```
learning_rate = knext.DoubleParameter("Lernrate", "Die Lernrate für
Adam.", 0.003, min_value=0.)
```

- ☐ Nutzung innerhalb der Ausführung Verfahren der Knotenklasse:

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

- `knext.StringParameter` für String-Parameter und Einzelauswahl:

- ☐ Unterschrift:

```
STringParameter(
    Label=None,
    Beschreibung=Nein,
    default_value=""
    enum: Liste[str] = Keine
    seit_version=Nein,
)
```

- ☐ Definition innerhalb einer Node/Parameter-Gruppenklasse:

```
# als Texteingabefeld
search_term = knext.StringParameter("Search term", "The string to search
im Text.", "

# als Einzelauswahl
Auswahl_param = knext.StringParameter("Selection", "Die Optionen zur Auswahl
aus.", "A", enum=["A", "B", "C", "D"])
```

- ☐ Nutzung innerhalb der Ausführung Verfahren der Knotenklasse:

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

- `knext.BoolParameter` für boolesche Parameter:

☐ Unterschrift:

```
knext.BoolParameter(  
    Label=None,  
    Beschreibung=Nein,  
    default_value=False,  
    seit_version=Nein,  
)
```

☐ Definition innerhalb einer Node/Parameter-Gruppenklasse:

```
output_image = knext.BoolParameter("Bilddarstellung aktivieren", "Option zur Ausgabe  
die Knotenansicht als Bild.", False)
```

☐ Nutzung innerhalb der Ausführung Verfahren der Knotenklasse:

```
wenn self.output_image wahr ist:  
# erzeugen Sie ein Bild der Handlung
```

- `knext.ColumnParameter` für eine einzelne Spaltenauswahl:

☐ Unterschrift:

```
knext.ColumnParameter(  
    Label=None,  
    Beschreibung=Nein,  
    port_index=0, # der Port, von dem aus die Eingabetabelle  
    Spalte_filter: Callable[[knext.Column], bool] = None, # a (lambda)  
    Funktion zu Filterspalten  
    beinhalten_row_key=False, # ob die Tabelle Row ID Spalte in  
    die Liste der wählbaren Spalten  
    include_none_column=False, # ob Sie None als wählbar aktivieren  
    Option, die "" zurückgibt  
    seit_version=Nein,  
)
```

☐ Definition innerhalb einer Node/Parameter-Gruppenklasse:

```
ausgewählt_col = knext.ColumnParameter(
    "Target Spalte",
    "Die Spalte mit Ländercodes auswählen."
    Spalte_filter = Lambda col: Stimmt, wenn "Land" in Col. Name sonst False,
    enthalten_row_key=False,
    enthalten_none_column=True,
)
```

- ☐ Nutzung innerhalb der Ausführung Verfahren der Knotenklasse:

```
wenn self.selected_column != "":
    Spalte = input_table[self.selected_column]
    # do something with the Spalte
```

- `knext.MultiColumnParameter` für eine Mehrfachspaltenauswahl

- ☐ Unterschrift:

```
knext.MultiColumnParameter(
    Label=None,
    Beschreibung=Nein,
    port_index=0, # der Port, von dem aus die Eingabetabelle
    Spalte_filter: Callable[[knext.Column], bool] = None, # a (lambda)
    Funktion zu Filterspalten
    seit_version=Nein,
)
```

- ☐ Definition innerhalb einer Node/Parameter-Gruppenklasse:

```
selekt_columns = knext.MultiColumnParameter(
    "Filterspalten",
    "Select die Spalten, die herausgefiltert werden sollten."
)
```

- ☐ Einrichtung innerhalb der Konfiguration Verfahren der Knotenklasse:

```
# der Mehrfachspaltenauswahlparameter muss die Liste von
Spalten einer Eingangstabelle
self.selected_columns = input_schema_1.column_names
```

- ☐ Nutzung innerhalb der Ausführung Verfahren der Knotenklasse:

```
für col_name in self.selected_columns:
    # Drop the Spalte aus der Tabelle
```

Alle oben genannten Argumente haben `Etiketten` und `Beschreibung`, die im Knoten angezeigt werden

Beschreibung in der KNIME Analytics Platform sowie im Konfigurationsdialog selbst.

Zusätzlich haben alle Parameterklassen ein optionales Argument `seit_version`, die

verwendet, um die Version der Erweiterung anzugeben, in der der Parameter eingeführt wurde. Bitte!

auf die [Ausführung Ihrer Erweiterung](#page35) Abschnitt unten für eine detailliertere Übersicht.

Parameter werden in Form von Klassenattributen innerhalb der Knotenklassendefinition definiert

(ähnlich zu Python) [Deskriptoren](#):

```
@knext.node(    ... )
...
Klasse MyNode:
num_repetitionen = knext.IntParameter(
    Label="Anzahl der Wiederholungen",
    Beschreibung="Wie oft eine Aktion wiederholen",
    default_value=42
)

def konfigurieren (    ... :
    ...

Def ausführen(    ... :
    ...
```

Während jeder oben aufgeführte Parametertyp eine Standardtypvalidierung hat, unterstützen sie auch die benutzerdefinierte

Validierung über eine immobilienähnliche Dekoration. Durch Umwickeln einer Funktion, die eine

vorläufiger Parameterwert, und erhöht eine Ausnahme sollte eine Bedingung verletzt werden, mit

die `@some_param.validator` dekorator, können Sie eine zusätzliche Ebene der Validierung hinzufügen

der Parameter `einige_param`. Dies sollte geschehen unten die Definition des Parameters für

die Sie ein Validierungsgerät hinzufügen, und oben die Konfiguration und Ausführung Methoden:

```
@knext.node(    ... )
...
Klasse MyNode:
num_repetitionen = knext.IntParameter(
Label="Anzahl der Wiederholungen",
Beschreibung="Wie oft eine Aktion wiederholen",
default_value=42
)

@num_repetitionen.validator
def validate_reps(Wert):
wenn Wert > 100:
ValueError("Zu viele Wiederholungen!")

def konfigurieren (    ... :
    ...

Def ausführen(    ... :
    ...
```

## Parameter Sichtbarkeit

Standardmäßig ist jeder Parameter eines Knotens im Konfigurationsdialog des Knotens sichtbar. Parameter kann als erweitert durch Einstellung gekennzeichnet werden `is_advanced = True`, die sie nur einmal zeigen der Benutzer hat im Konfigurationsdialog „Weitere Einstellungen anzeigen“ geklickt.

Manchmal sollte ein Parameter nur für den Benutzer sichtbar sein, wenn ein anderer Parameter einen bestimmten Parameter aufweist. Wert. Dazu weist jeder oben aufgeführte Parametertyp ein Verfahren auf Regel. Bei diesem Verfahren kann man eine auf einem anderen Parameter basierende Bedingung angeben und die darauf anzuwendende Wirkung Parameter, wenn die Bedingung wahr wird.

```
@knext.node(args)
Klasse MyNode:
string_param = knext.StringParameter(
"String Param Title",
"String Param Title Beschreibung",
"Standardwert"
)

# dieser Parameter wird deaktiviert, wenn string_param "foo" oder "bar" ist
int_param = knext.IntParameter(
"Int Param Title",
"Int Param Beschreibung",
).rule(knext. OneOf(string_param, ["foo", "bar"]), knext. Effekt.DISABLE)
```

[

Derzeit unterstützt dies nur Bedingungen, bei denen ein anderer Parameter genau entspricht einem Wert. Regeln können nur von Parametern auf der gleichen Ebene abhängen, nicht in einer Kinder- oder Elternparametergruppe.

Die vollständige API-Dokumentation der [Regel](#) [Verfahren](#) [Hier](#) .

## Parametergruppen

Es ist auch möglich, Gruppen von Parametern zu definieren, die als separate Abschnitte angezeigt werden im Konfigurationsdialog UI. Durch die Verwendung `@knext.parameter_group` Dekorator mit einem [Datenklasse](#) - wie Klassendefinition, Sie sind in der Lage, Parameter und, optional, ihre Validatoren in eine separate Einheit außerhalb der Node-Klasse-Definition, halten Ihren Code sauber und pflegefähig. Eine Parametergruppe ist ebenso wie ein einzelner Parameter mit einem Knoten verknüpft wäre:

```
@knext.parameter_group(label="Meine Einstellungen")
Klasse MySettings:
    Name = knext.StringParameter("Name", "Der Name der Person", "Bario")

    num_repetitionen = knext.IntParameter("NumReps", "Wie oft wiederholen wir?", 1,
    min_value=1)

    @num_repetitionen.validator
    def reps_validator(Wert):
        wenn Wert == 2:
            ValueError("Ich mag die Nummer 2") nicht

    @knext.node(    ... )
    ...
    Klasse MyNodeWithSettings:
        Einstellungen = MySettings()

    def konfigurieren (    ... :
        ...

    Def ausführen(    ... :
        ...
        Name = selbst.settings.name
        ...
```

Ein weiterer Vorteil der Definition von Parametergruppen ist die Fähigkeit zur Gruppenvalidierung. Wie nicht nur in der Lage zu sein, einen einzigen Wert zu validieren, wenn ein Validierungsgerät an einem Parameter, Gruppenvalidatoren haben Zugriff auf die Werte aller in der Gruppe, die eine komplexere Validierungsroutine ermöglicht.

Wir bieten zwei Möglichkeiten, einen Gruppenvalidator zu definieren, mit Werten das ist ein Wörterbuch der Parameter\_Name : Parameter\_Wert Mappings:

1. durch Durchführung einer validate(Selbst, Werte) Verfahren innerhalb der Parametergruppenklasse

Definition:

```
@knext.parameter_group(label="Meine Fraktion")
Klasse MyGroup:
    first_param = knext.IntParameter("Simple Int", "Testing a simple int
    Param", 42)
    second_param = knext.StringParameter("Simple String", "Testing a simple string
    param", "foo")

    def validate(Selbstwerte):
        wenn Werte["first_param"] < len(Werte["second_param"]):
            ValueError("Params sind unsymmetrisch!")
```

2. durch Verwendung der vertrauten @group\_name.validator Dekoration mit einem Validierungsgerät

Funktion innerhalb der Klassendefinition des "parent" der Gruppe (z.B. des Knotens selbst, oder eines verschiedene Parametergruppe:

```
@knext.parameter_group(label="Meine Fraktion")
Klasse MyGroup:
    first_param = knext.IntParameter("Simple Int", "Testen eines einfachen Int
    Param", 42)
    second_param = knext.StringParameter("Simple String", "Testing a simple string
    param", "foo")

    @knext.node( ... )
    ...
    Klasse MyNode:
        param_group = MyGroup()

        @param_group.validator
        def validate_param_group(Werte):
            wenn Werte["first_param"] < len(Werte["second_param"]):
                ValueError("Params sind unsymmetrisch!")
```

Wenn Sie einen Validierungsprüfer mit der ersten Methode definieren und dann eine andere definieren Validator für die gleiche Gruppe mit dem zweiten Verfahren, der zweiten

validator wird **Overrid** der erste Validator. Wenn Sie möchten zu halten **beide** validatoren aktiv, können Sie die optionale Überschreiben = False Argumentation der Dekorator: @param\_group.validator(override=False) .

Intuitiv können Parametergruppen in andere Parametergruppen geschachtelt werden, und deren

Parameterwerte, die während der Validierung der Elterngruppe zugegriffen wurden:

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

@knext.parameter_group(label="Outer Group")
Klasse OuterGroup:
äußere_int = knext.IntParameter("Outer Int", "The äußere int param", 2)
InnerGroup()

def validate(Selbstwerte):
wenn Werte["inner_group"]["inner_int"] > Werte["outer_int"]:
heben ValueError("Der innere Int sollte nicht größer sein als der äußere!")
```

## Erklärung zur Nichteinhaltung

Sie können die `@knext.output_view(name="", Beschreibung="")` `@knext.output_view` dekorator, um anzugeben, dass ein node gibt eine Ansicht zurück. In diesem Fall `ausführung` Methode sollte eine Tupel von Port-Ausgängen zurückgeben und die Ansicht (von Typ `knime.api.views.NodeView` )

```

aus der Einfuhr Liste
import knime. Verlängerung als knext
Import seaborn as sns

@knext.node(name="My Node", node_type=knext.NodeType.VISUALIZER, icon_path="icon.png",
Kategorie="/")
@knext.input_table(name="Input Data", Beschreibung="Wir lesen Daten von hier")
@knext.output_view(name="Meine schöne Aussicht", Beschreibung="Darsteller für Meer")
Klasse MyViewNode:
"""
Ein Aussichtsknoten

Dieser Knoten zeigt ein Diagramm.
"""

def konfigurieren(selbst, config_context, input_table_schema):
Reisepass

def ausführen(self, exec_context, table):
df = Tabelle.to_pandas()
ns.lineplot(x="x", y="y", data=df)
zurück knext.view_seaborn()

# Wenn die Knotentabellen ausgibt, muss die Ausgangsansicht
# be the last element of the return value
#
# output_table = knext.from_pandas(df)
# return output_table, knext.view_seaborn()
#
# Für mehrere Tabellenausgänge verwenden
# return output_table_1, output_table_2, knext.view_seaborn()

```

## Zugriff auf Durchflussgrößen

Sie können auf die Flussvariablen zugreifen, die dem Knoten in beiden **Konfiguration** und **Ausführung** Methoden, über die `config_context.flow_variables` und `exec_context.flow_variables` Attribute. Die Strömungsgrößen sind als Wörterbuch der `config_context` Dateiname: `config_context.flow_variables["Dateiname"]` : Der Wert von `exec_context.flow_variables["Dateiname"]` Mappings und unterstützen folgende Typen:

- `bool`
- `Liste(bool)`
- `Flott`
- `Liste(Flott)`

- int
- Liste(int)
- Str
- Liste(str)

Durch Mutation der `strom_variables` Wörterbuch, Sie können auf, ändern und löschen vorhandener Fluss Variablen sowie neue zu verbreitende, nachgeschaltete Knoten erstellen.

## Ausführung Ihrer Erweiterung

Während Sie Ihre Erweiterung nach der ersten Veröffentlichung weiter entwickeln, können Sie die Funktionalität Ihrer Knoten durch Hinzufügen oder Entfernen bestimmter Parameter. Mit der Versionierung Fähigkeiten von Python-basierten Knotenerweiterungen für die KNIME Analytics Platform, können Sie sicherstellen Rückwärtskompatibilität für Ihre Benutzer.

Wie in der [Python Node Erweiterung](#page11) Abschnitt, die `knime.yml` Konfigurationsdatei enthält `Version` Feld. Dies ermöglicht es Ihnen, jeder Iteration Ihres Erweiterung. Wie genau wollen Sie dem folgen [semantische Version](#) System ist komplett auf an Sie, aber wir erfordern die Einhaltung der folgenden Formatierungs-Regel: Versionen müssen bestehend aus drei nicht-negativen numerischen, durch Punkte getrennten Teilen (z. `1.0.0`, `0,21`, usw.).

Die Versionsnummern werden von links nach rechts, d.h. verglichen. `1.0.1` ist neuer als `1.0.0`, aber älter als `1.1.0`.

Beim Hinzufügen eines neuen Parameters zu einem Knoten sollten Sie ihn mit dem entsprechenden Version Ihrer Erweiterung. Dies geschieht mit `seit_version` Argumentation, die jetzt über den entsprechenden Konstruktor (z. `knext.IntParameter`), sowie Parametergruppen über die `@knext.parameter_group` dekorator. Wenn nicht angegeben, `seit_version` Ein Argument einer Parameter- oder Parametergruppe setzt voraus, `0,0.0`, die zeigt an, dass der Parameter aus der ersten Iteration der Erweiterung zur Verfügung stand.

Ein häufiger Anwendungsfall der Erweiterungsversion ist, die Rückwärtskompatibilität zu erleichtern, wenn Eröffnung von Workflows, die mit einer älteren Version der installierten Erweiterung erstellt/erreicht wurden auf der Maschine. Was ist das? Analytics-Plattform wird versuchen, in diesem Fall standardmäßig zu erreichen, ist die Werte der zuvor konfigurierten Knoteneinstellungen, die noch in der aktuelle Version der Erweiterung mit den neu hinzugefügten Knoteneinstellungen, falls vorhanden. Letztere sind dann automatisch auf ihre Standardwerte eingestellt und der Knoten bleibt konfiguriert.

[

Manchmal sollte der Standardwert für einen neu hinzugefügten Knoten anders sein als der Standardwert für einen Knoten, der als Teil eines alten Workflows geladen wird (für einen Beispiel siehe [Doppel-Parameter](#) unten). In diesem Szenario können Sie `DefaultValueProvider` anstatt des Standardwerts. Die `DefaultValueProvider` ist eine Funktion, die `Version` den Standardwert des Parameters erzeugt für die Version der Erweiterung. Für alte Workflows wird es mit dem Erweiterungsversion der Workflow wurde mit gespeichert. Für neue Workflows wird es genannt mit der aktuellen Version der Erweiterung.

Hier ist ein minimales Funktionsbeispiel einer Python-basierten Erweiterung mit einem einzigen Knoten mit einem einzigen Parameter. Da der Parameter aus der anfänglichen Freigabe der Erweiterung, können wir die Einstellung `seit_version` Argument:

```
"""
Meine Extension | Version: 0.1.0 | Autor: Jane Doe
"""

import knime. Verlängerung als knext

@knext.node(
    "Meine Node",
    knext.NodeType. GERICHTSHOF
    "..icons/icon.png",
    "/"
)
@knext.output_table("Output Data", "Daten, die von diesem Knoten generiert werden")
Klasse MyNode:
    """Short Node Beschreibung.
    Lange Knotenbeschreibung.
    """
    my_param = knext.IntParameter(
        "Mein Param",
        "Mein Int-Parameter."
        42,
    )

    def konfigurieren(selbst, config_context, input_table_schema):
        zurückgeben_table_schema

    def ausführen(self, exec_context, input_table):
        df = input_table.to_pandas()
        df['column1'] += Selbst.my_param
        Zurück knext. Tabelle.from_pandas(df)
```

Während der nächsten Versionen der Erweiterung, `MyNode` mit einem Zusatz von mehreren modifiziert neue Parameter:

```

"""
Meine Extension | Version: 0.5.0 | Autor: Jane Doe
"""

import knime. Verlängerung als knext

@knext.node(
    "Meine Node",
    knext.NodeType. GERICHTSHOF
    "..icons/icon.png",
    "/"
)
@knext.output_table("Output Data", "Daten, die von diesem Knoten generiert werden")
Klasse MyNode:
"""Short Node Beschreibung.
Lange Knotenbeschreibung.
"""

my_param = knext.IntParameter(
    "Mein Param",
    "Mein Int-Parameter."
    42,
)
doppel_param = knext.DoubleParameter(
    "My Double",
    "Doppel-Parameter, der versucht, Pi zu sein.",
    # Für alte Workflows muss der Wert 1 sein, um rückwärtskompatibel zu bleiben
    # but for new workflows we want the default to be 3.14
    Lambda v: 1 wenn v < knext.Version(0, 3, 0) andere 3.14,
    seit_version="0.3.0",
)
string_param = knext.StringParameter(
    "My String",
    "Ein wichtiger String-Parameter, der in eine Flussvariable verwandelt werden soll."
    "Foo",
    da_version="0,5.0",
)

def konfigurieren(selbst, config_context, input_table_schema):
    zurückgeben_table_schema

def ausführen(self, exec_context, input_table):
    df = input_table.to_pandas()
    df['column1'] += selbst.my_param * self.double_param
    exec_context.flow_variables['important_string'] = self.string_param
    Zurück knext. Tabelle.from_pandas(df)

```

Nun, wenn ein Benutzer, dessen Version von `Meine Erweiterung` ist `0,5.0` öffnet einen Workflow, der `MyNode` die auf einer Maschine, in der die Version `Meine Erweiterung` war, für Beispiel: `0,20` , die Knoteneinstellungen werden automatisch angepasst, um die zuvor

konfigurierter Wert für `My_param` , und die Standardwerte für `Doppel-Parameter` und `string_param` . wenn der Benutzer sollte den Knoten ausführen, ohne ihn erst neu zu konfigurieren, `Ausführung Methode` würde diese Standardwerte für die entsprechenden Parameter verwenden.

Beachten Sie, wie der Standardwert `Doppel-Parameter` hängt von der Version ab, um sicherzustellen, dass die Ausgabe des Knotens ändert sich nicht, wenn der Workflow einer älteren Version ist.

Hat sich das Verhalten/Funktionalität des Knotens während der verschiedenen Versionen der Erweiterung, und Sie möchten, dass Benutzer den Knoten neu konfigurieren, wenn bestimmte Bedingungen Sie können die `config_context.set_warning()` oder `exec_context.set_warning()` Methoden in der `Konfiguration` und `Ausführung Methoden` Ihres Knotens bzw. zur Anzeige eines gelben "Warning"-Zeichen im Knotenstatus. Zusätzlich können Sie eine Ausnahme erhöhen, um die Benutzer, um den Knoten neu zu konfigurieren. Zum Beispiel:

```

import knime. Verlängerung als knext

@knext.node(
    "Meine Node",
    knext.NodeType. GERICHTSHOF
    "..icons/icon.png",
    "/"
)
@knext.output_table("Output Data", "Daten, die von diesem Knoten generiert werden")
Klasse MyNode:
    """Short Node Beschreibung.
    Lange Knotenbeschreibung.
    """
    my_param = knext.IntParameter(
        "Mein Param",
        "Mein Int-Parameter."
        42,
    )
    doppel_param = knext.DoubleParameter(
        "My Double",
        "Doppel-Parameter, der versucht, Pi zu sein.",
        Lambda v: 1 wenn v < knext.Version(0, 3, 0) andere 3.14,
        seit_version="0.3.0",
    )

    def konfigurieren(selbst, config_context, input_table_schema):
        wenn selbst.my_param < 10:
            config_context.set_warning("Bitte rekonfigurieren Sie den Knoten.")
            ValueError("Mein Param kann nicht weniger als 10 sein.")

        zurückgeben_table_schema

    def ausführen(self, exec_context, input_table):
        df = input_table.to_pandas()
        df['column1'] += selbst.my_param * self.double_param
        Zurück knext. Tabelle.from_pandas(df)

```

## Abgrenzung von Knoten

Manchmal ist es nicht möglich, einen Knoten zu ändern und rückwärtskompatibel zu bleiben, z.B. wenn ein Ein- oder Ausgabeport wird hinzugefügt. Wenn Sie sich in diesem Szenario finden, tun Sie die folgenden:

- Deprecate den alten Knoten, indem Sie die `is_depreciert` Argumentation wahr in der `knime.extension.node` dekorator. Der Knoten wird dann im Knoten nicht mehr aufgeführt Repository, kann aber noch in bestehenden KNIME-Workflows geladen werden, in denen es dann auch als abgeschrieben gekennzeichnet.

- Implementieren Sie eine neue Version des Knotens, der die gleiche Name Argument in der  
`knime.extension.node` dekorator als der alte Knoten.

Ändern Sie nicht den Namen der Python-Klasse, die Ihren alten Knoten implementiert  
weil dieser Name als ID der Analytics Plattform verwendet wird, um den Knoten zu finden.

Verbesserung der Knotenbeschreibung mit Markdown

Die Beschreibung Ihres Knotens, der in der Warenbezeichnung Bereich von KNIME Analytics  
Plattform, wenn ein Knoten ausgewählt wird, besteht aus mehreren Komponenten. Diese Komponenten  
kommen aus den Beschreibungen, die Sie, als Entwickler, bei der Definition der Bausteine  
B. des Knotens, wie die Eingangsports oder die Konfigurationsparameter.

Denken Sie daran, dass in der ersten Zeile der Beschreibung docstring, neben den drei  
doppelte Zitate, Sie können eine kurze Beschreibung, die in der  
Übersicht beim Anklicken einer Kategorie im Knoten-Repository des KNIME  
Analyseplattform.

Einschließlich Aufschlüsselung Python Paket im Python Umgebung mit Ihrem  
Knotenerweiterung, können Sie nutzen Markiert Syntax beim Schreiben dieser Beschreibungen  
Verbesserung der Lesbarkeit und des Gesamtausdrucks der Dokumentation Ihrer Knoten.

Hier finden Sie eine Liste, auf der Markdown-Syntax für jede Knotenbeschreibung unterstützt wird.  
Element.

Als KNIME Analytics Plattform Moderne UI , wir werden an  
erweitert unsere Unterstützung für zusätzliche Markdown-Syntax.

Tabelle 1. Die unterstützte Markdown-Syntax für die verfügbaren Knotenbeschreibungskomponenten

Element	Node Beschreibung	Hafenbeschreibung	Parameter Beschreibung	Top-Level Parameter Gruppe Beschreibung
<u>Bezeichnung</u>	✓	✗	✗	✗
<u>Bolzen</u>	✓	✓	✓	✓
<u>Italic</u>	✓	✓	✓	✓

Element	Node Beschreibung	Hafenbeschreibung	Parameter Beschreibung	Top-Level Parameter Gruppe Beschreibung
<a href="#">Bestellliste</a>	✓	✓	✓	✓
<a href="#">Ungeordnete Liste</a>	✓	✓	✓	✓
<a href="#">Code</a>	✓	✓	✓	✓
<a href="#">Fen-Code Blöcke</a>	✓	✓	✓	✗
<a href="#">Regelmäßig</a>	✓	✓	✗	✗
<a href="#">Link</a>	✓	✓	✓	✓
<a href="#">Tabelle</a>	✓	✗	✗	✗

Hier ist ein funktionelles Beispiel für die Verwendung von Markdown beim Schreiben eines Python-Knotens:

```
import knime. Verlängerung als knext

@knext.parameter_group("Keine Einstellungen")
Klasseneinstellungen:
"""
Einstellungen, um zu konfigurieren, wie der Knoten mit den bereitgestellten **JSON*-Strings arbeiten soll.
"""

Klasse LoggingOptionen(knext.EnumParameterOptionen):
NONE = ("None", "Logging *disabled*.")
INFO = ("Info", "Allow *some* protokolliert angezeigt werden.")
VERBOSE = ("Verbose", "Log *everything*.")

logging_verbosity = knext.EnumParameter(
    "Einloggen der Verbenheit",
    "Setze den Knoten, der die Verbenheit während der Ausführung protokolliert."
    LoggingOptionen.INFO.name,
    LoggingOptionen,
)
```

```

discard_missing = knext.BoolParameter(
    "Discardzeilen mit fehlenden Werten",
    """
Verwenden Sie diese Option, um Zeilen mit fehlenden Werten zu verwerfen.

- Wenn enabled, wird der Knoten Zeilen ignorieren, in denen ein Attribut des JSON
strings hat fehlenden Wert.

- Wenn disabled, wird der Knoten solche Zeilen mit den entsprechenden fehlenden
Werte.

" "
,
Stimmt.

)

@knext.node("JSON Parser", knext.NodeTyp.MANIPULATOR, "icon.png", main_category)
@knext.input_table(
    "Eingangstabelle",
    """
Eingabetabelle mit JSON-codierten Strings in jeder Zeile.

Beispielformat der erwarteten Eingabe:
`` `
{\\cHFFFF}

"Konstanz": {\\cHFFFF}

"Bevölkerung": 90.000,

"Region": "Baden-Württemberg",

...
},
...
}
`` `
" "
,
)

@knext.output_table(
    "Parsed JSON",
    "Ausgangstabelle mit Spalten mit den aus der angegebenen
JSON String.",
)

Klasse JsonParser:
"""Node for parsing JSON strings.

Bei einer Tabelle mit [JSON](https://developer.mozilla.org/en-US/docs/Glossary/JSON) Strings, dieser Knoten versucht, sie zu parsen und
gibt die extrahierten Informationen in einer neuen Tabelle aus.

| Erlaubt | Nicht erlaubt
-----|
| JSON | YAML |
"""

Einstellungen = Einstellungen()

def konfigurieren(selbst, config context, input table schema):

```

```
# Konfigurationsroutine
...
zurückgeben_table_schema

def ausführen(self, exec_context, input_table):
    # Ausführungsroutine
    ...
    Zurück zur Übersicht
```

Im Folgenden finden Sie die daraus resultierende Knotenbeschreibung in der KNIME Analytics Platform:

**JSON Parser**

Given a table containing **JSON** strings, this node attempts to parse them and outputs the extracted information in a new table.

Allowed	Not allowed
JSON	YAML

**Dialog Options**

**Node settings**

Settings to configure how the node should work with the provided **JSON** strings.

**Logging verbosity**

Set the node logging verbosity during execution.

**Available options:**

- None: Logging *disabled* .
- Info: Allow *some* logging messaged to be displayed.
- Verbose: Log *everything* .

**Discard rows with missing values**

Use this option to discard rows with missing values.

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

**Ports**

**Input Ports**

0 Input table containing JSON-encoded strings in each row.

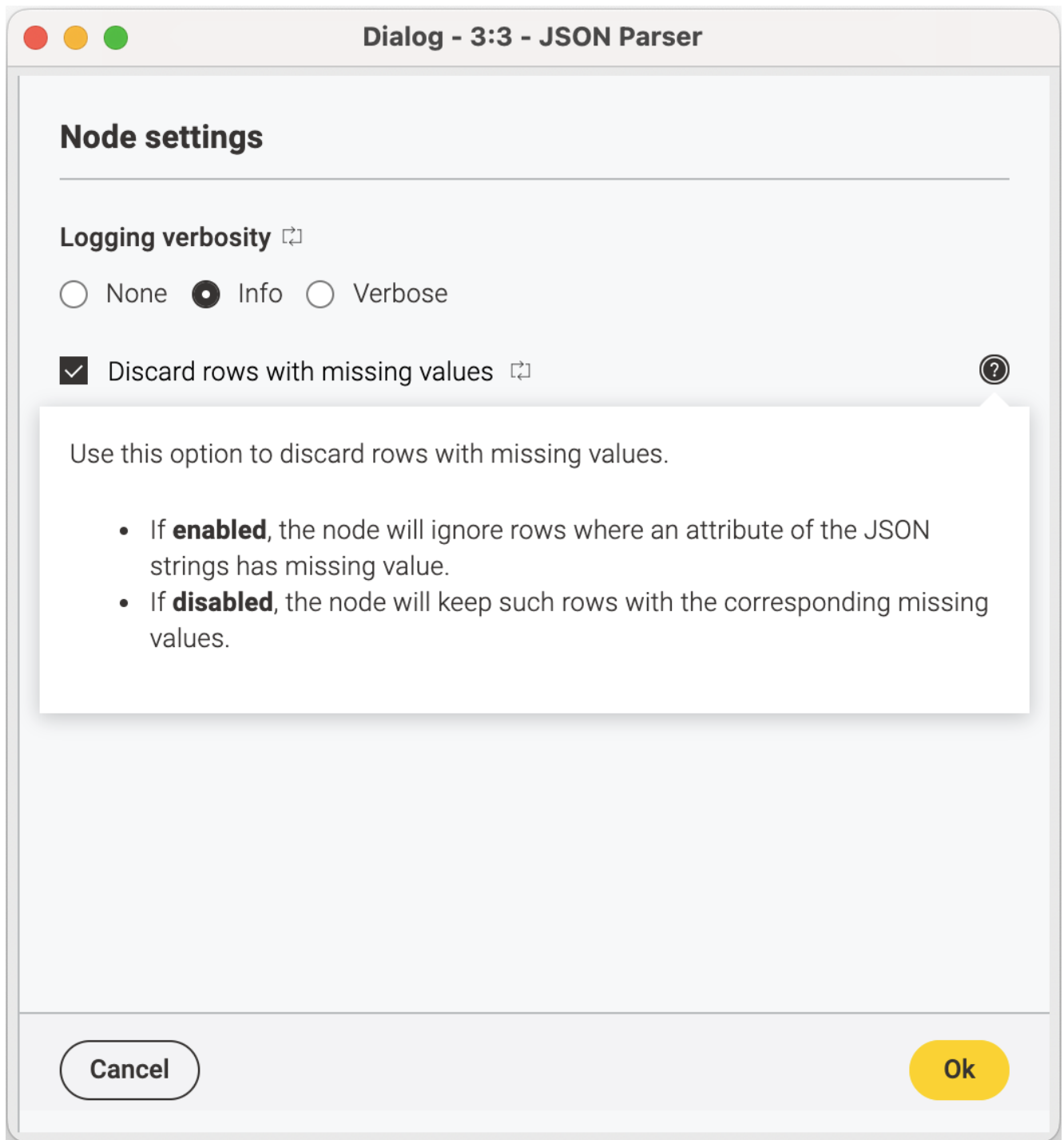
Example format of the expected input:

```
{
  "Konstanz": {
    "population": 90,000,
    "region": "Baden-Württemberg",
    ...
  },
  ...
}
```

**Output Ports**

0 Output table containing columns with the information extracted from the provided JSON string.

Die Beschreibungen einzelner Knotenparameter können zusätzlich innerhalb der Konfigurationsdialog des Knotens:



## Teilen Sie Ihre Extension

Sie können Ihre Erweiterung auf zwei Arten teilen. Man ist, die Erweiterung zu bündeln, um eine lokale aktualisieren Sie die Website, die mit Ihrem Team geteilt werden kann oder zum Testen verwendet werden. Der andere ist, es zu veröffentlichen auf KNIME Community Hub und machen es für die Gemeinschaft zur Verfügung. Entweder die beiden Optionen brauchen einige Setup-Details. In diesem Abschnitt wird das Setup und die beiden Optionen erläutert.

### Einrichtung

Um sicherzustellen, dass die Benutzer, die Sie Ihre Erweiterung mit geteilt haben, sind in der Lage, seine Nutzung Funktionalität vollständig und fehlerfrei, wir bündeln die Quelldateien zusammen mit den erforderlichen Pakete mit `Pixi-Pack`.

Die `knime.yml` Datei (Referenz zum [Python Node Erweiterung Setup](#) Abschnitt für ein Beispiel hierfür Konfigurationsdatei) enthält die zur Bündelung Ihrer Extension erforderlichen Informationen, einschließlich:

- `Erweiterung_Modul` : Name des `.py` Datei, die die Knotendefinitionen Ihres Erweiterung.
- `Pixi_toml_path` : der Weg zum `Pixi.toml` die Konfiguration der Python Umgebung, die mit Ihrer Erweiterung verwendet wird (siehe Beispiel unten).

Die Vorlage ist in [knime-python-extension-templat](#) enthält bereits `Pixi.toml` Datei mit einer Basis Python Umwelt. Sie können diese Datei als Ausgangspunkt für sich selbst verwenden Erweiterung. Sie können zusätzliche Pakete zum `Pixi.toml` Datei mit dem Befehl:

```
Pixi add
```

Dies wird das Paket automatisch in das Paket einfügen `Pixi.toml` und stellen Sie sicher, dass die Umwelt funktioniert auf allen Plattformen. Pakete von PyPI ( `Pip` ) kann auch durch Verwendung hinzugefügt werden Der Befehl `pixi add --pypi` . Es sei jedoch darauf hingewiesen, dass `Pip...` Quelle Pakete werden nicht in der Bündelung pipeline unterstützt. Stellen Sie sicher, nur conda enthalten Pakete (bevorzugt) oder Pypi-Radpakete. Wenn Sie keine Pip-Source-Pakete verwenden können, Sie können Ihr eigenes conda-Paket erstellen und es auf [anaconda.org](#) oder einer privaten conda hosten Kanal. Bevor Sie ein Paket als `Pypi` Abhängigkeit, stellen Sie sicher, ob es verfügbar ist auf [conda-forge](#) oder anderen conda-Kanälen. Wenn es ist, bevorzugen Sie die Verwendung des conda-Pakets.

Wenn Sie bereits einen `Conda` Umgebung mit den gewünschten Paketen, können Sie die Befehl:

```
conda env export --from-history > env.yml
```

zur Erzeugung einer `env.yml` eine Datei, in der die in der Umgebung installierten Pakete aufgelistet sind. Das wird die Liste der Abhängigkeiten auf die Pakete reduzieren, die Sie manuell installiert haben die Umwelt. Beachten Sie, dass diese Option nicht die Liste der manuell spezifizierten Kanäle bei der Installation von Paketen (z. `conda-forge`), so müssen Sie sie hinzufügen selbst. Sie können diese Datei dann als Ausgangspunkt für Ihre `Pixi.toml` Datei. Verwendung:

```
Pixi init -i
```

Sie können eine `Pixi.toml` Datei, die die in der `env.yml` Datei. Anmerkung jedoch, dass diese Datei nicht die Basis enthalten `Python` Umgebung erforderlich für Ihre Erweiterung, wenn Sie es manuell hinzufügen. Sie können jedoch die Abhängigkeiten der beiden zusammenführen `Pixi.toml` Dateien.

`Pixi.toml` :

```
(Arbeitsraum)
Autoren = ["Bobby Test "]
Kanäle = ["knime", "conda-forge"]
Name = "my-knime-python-extension"
Plattformen = ["win-64", "linux-64", "osx-64", "osx-arm64"]
Version = "0.1.0"

(tasks)

(abhängigkeiten)
python = "3.11.*" # Basisabhängigkeit
knime-extension = "{version_ap}" # Basisabhängigkeit
knime-python-base = "{version_ap}" # Basisabhängigkeit
scipy = "*" # Beispielabhängigkeit, die vom Benutzer für den
Verlängerung

[feature.build.dependencies]
python = "3.9.*" # Abhängigkeit für den Bündelungsprozess
knime-extension-bundling = "{version_ap}"
Verfahren

(feature.build.tasks)
build = { args = [{ "arg" = "dest", "default" = "./local-update-site" }], cmd = "python
$CONDA_PREFIX/bin/build_python_extension.py . {{ dest }}" } # Befehl zur Bündelung des
Verlängerung

(Umwelt)
build = { features = ["build"], no-default-feature = true}
bündeln die Erweiterung
```

[

Sie können immer überprüfen, ob eine Umgebung richtig definiert ist, ohne sie zu installieren indem Sie den Befehl ausführen `pixi schloss` im Terminal. Dies wird eine `Pixi.lock` Datei im gleichen Verzeichnis wie die `Pixi.toml` Datei, die Versionen der Pakete, die in der Umgebung verwendet werden. Diese Datei kann verwendet werden, um die Umwelt auf anderen Maschinen oder um es mit anderen Benutzern zu teilen und wird verwendet während des Bündelungsprozesses. Wenn `Pixi.lock` Datei wird nicht erstellt, es bedeutet, dass es gibt einen Fehler in der `Pixi.toml` Datei. Die Fehlermeldung wird in der Terminal. Pixi wird versuchen, die Abhängigkeiten für alle in der `Pixi.toml`, die `Win-64`, `linux-64`, `osx-64`, und `osx-arm64`, wie Erweiterungen sollten in der Regel für alle Plattformen verfügbar sein. Sie können aber auch verschiedene Umgebungen für jede Plattform durch die Nutzung der Plattformen Argument beim Hinzufügen eines Pakets.

Nach der Einstellung `Pixi.toml` Datei, Sie können den Befehl ausführen `Pixi installieren` zur Installation Pakete in der `Pixi.toml` Datei. Dies wird eine neue Umgebung in der `.pixi/envs/default` Verzeichnis. Sie können den Pfad zu diesem Verzeichnis direkt als Python verwenden Pfad (z.B. im VS-Code) um Ihre Erweiterung lokal auszuführen. In der Einstellung "Select Python Interpreter" klicken Sie auf "Enter Dolmetschpfad" und wählen Sie den Python ausführbar in der `.pixi/envs/default` Verzeichnis. Damit können Sie die Linter- und Autokomplete-Funktionen nutzen Ihre IDE. Alternativ können Sie den Befehl verwenden `Pixischale` die Umwelt "aktivieren" ähnlich `conda aktivieren`.

## OS-spezifische Umgebungen

Seit KNIME Analytics Platform ist unter Windows, Linux und macOS verfügbar, sollten Sie versuchen Ihr Bestes, um sicherzustellen, dass Ihre Python Erweiterung wie erwartet auf allen Plattformen. Ideal, durch Verwendung `Conda` Pakete für alle Plattformen. Dies ist jedoch nicht immer möglich, und einige Pakete sind nur für bestimmte Betriebssysteme verfügbar. In diesem Fall brauchen Sie verschiedene plattformspezifische Pakete in der `Pixi.toml` Datei. Dies kann einfach gemacht werden durch die Verwendung `Plattformen` Argument beim Hinzufügen eines Pakets:

```
Pixi add -p
```

wenn ist eine gemeinschaftlich getrennte Liste von Plattformen, für die das Paket während der bündeln Prozess, die `Pixi` Werkzeug wird automatisch das richtige Paket für jede Plattform verwenden, und die richtige Umgebung wird für jede Plattform erstellt.

Unterstützung für Apple Silicon-spezifische Umgebungen ist ab der  
4.7 Veröffentlichung der KNIME Analytics Platform. Der Name der conda Plattform ist  
Arm64 .

Schließlich braucht eine neue Erweiterung eine LICENSE.TXT die während der Installation angezeigt wird  
Prozess.

Option 1: Bündeln einer Python-Erweiterung, um eine Reißverschluss-Update-Seite zu teilen

Sobald Sie die Implementierung Ihrer Python-Erweiterung abgeschlossen haben, können Sie sie zusammen mit  
gegebenenfalls Python Umwelt, in eine lokale Update-Website. Dies ermöglicht anderen Benutzern die Installation  
Ihre Erweiterung in der KNIME Analytics Platform.

[Sobald Sie die von Ihnen verwendete Umgebung vorbereitet haben](#)

Erweiterung (mit einem korrekten und abschließbaren pixi.toml) und haben die knime.yml Datei, Sie  
kann die lokale Update-Website generieren.

Im pixi.toml haben wir einen dedizierten Build-Befehl für die Bündelung Ihrer Erweiterung vorbereitet.  
Dieser Befehl erstellt automatisch eine Python-Umgebung mit allen Abhängigkeiten  
notwendig für die Bündelung. Einfach den Befehl ausführen:

```
Pixi run build dest =
```

wenn Des ist das Zielverzeichnis, in dem die Update-Website erstellt wird. um dich zu bündeln  
Erweiterung, wo ist der Pfad zum Verzeichnis, in dem die gebündelte Erweiterung  
die aktualisierte Website wird erstellt. Der Befehl erstellt eine Umgebung mit dem Namen Bau und  
alle in der Pixi.toml Datei, einschließlich der knime-extension-  
Füllung Paket. Dieses Paket enthält die notwendigen Werkzeuge, um automatisch Ihre  
Erweiterung.

Unter der Haube wird dieser Befehl die Bau Umwelt (wenn nicht schon) und die  
Befehl build\_python\_extension . in der Pixi Shell. Die . Angaben  
das aktuelle Verzeichnis, das die knime.yml Datei. Die ist der Weg zu  
das Verzeichnis, in dem die Update-Website erstellt wird. build\_python\_extension ist ein Python  
Skript, das in der Conda Paket knime-extension-Füllung . Für die meisten Anwendungsfälle,  
es genügt, diesen Befehl auszuführen, um Ihre Erweiterung zu bündeln. Die  
build\_python\_extension.py Skript kann auch direkt ausgeführt werden, so dass zusätzliche Optionen  
werden übergeben. Alle verfügbaren Optionen finden Sie durch den Befehl Python  
build\_python\_extension.py --help im Terminal (mit der Build-Umgebung aktiviert).

[

Standardmäßig bündelt das Skript die Erweiterung für die neuesten KNIME Analytics Plattformversion. Wenn Sie die Erweiterung für ein bestimmtes KNIME bündeln möchten Version, Sie müssen das entsprechende conda-Paket installieren. Sie können angeben die Version, wenn Sie die Umgebung erstellen, z. `knime-extension-bundling=5.7`. Bei der Bündelung einer älteren Version, die Umgebung TOML-Dateien **müssendie** entsprechenden Versionen der `knime-python-Basis` und `knime-Verlängerung` Pakete, z. `- knime-python-base = 5.6` bei Bündelung der Version 5.6.

[

Der Bündelungsprozess kann mehrere Minuten dauern. Ein schnelles Internet Eine Verbindung ist vorteilhaft.

Hinzufügen der generierten **Repository** Ordner zur KNIME Analytics Platform als Software-Site in [Datei → Vorlieben](#) → [Installieren und aktualisieren](#) → [Verfügbare Software-Sites](#)

Endlich, installieren Sie es über [Datei → KNIME installieren Erweiterungen](#)

Das generierte Repository kann nun von anderen Benutzern geteilt und installiert werden.

## Option 2: Veröffentlichen Sie Ihre Erweiterung auf KNIME Community Hub

Sobald Sie die Implementierung Ihrer Python-Erweiterung abgeschlossen haben, können Sie es gemeinsam mit gegebenenfalls `Python` Umwelt, auf KNIME Community Hub. Dies ermöglicht anderen Benutzern leicht entdecken, installieren und verwenden Sie Ihre Erweiterung.

Bieten Sie die Erweiterung an

[<a href="#page45" style="color: #2e3192; text-decoration: underline;">Erweiterung</a> \[<a href="#page45" style="color: #2e3192; text-decoration: underline;">Erweiterung</a> \\[<a href="#page45" style="color: #2e3192; text-decoration: underline;">Erweiterung</a>\\]\\(#\\)\]\(#\)](#)

Umwelt und `knime.yml` eine Datei mit Metadaten über Ihre Extension.

Laden Sie Ihre Erweiterung auf ein Git-Repository hoch, um sicherzustellen, dass die `knime.yml` Datei befindet sich an der obere Ebene des Projektarchivs. A `config.yml` Datei wird nicht für das Verlagswesen benötigt.

Hier ist eine empfohlene Projektstruktur:

```
https://github.com/user/my_knime_extension
.
Icons
Ψ ψ ⚡ icon.png
ÄÖ å â ã icon.png
Ψ ⚡ – Erweiterung. Hefe
ÄÖ å â ã icon.png
Ψ ψ ⚡ — Beispiel_mit_Python_node.knwf
ÄÖ å â ã icon.png knime.yml
- Pixi.toml
ÄÖ å â ã ⚡ config.yml # nicht zum Veröffentlichen benötigt
LICENSE.TXT
⚡ — README.md
```

## Schreibe einen Test-Workflow

ANHANG Installieren Sie die KNIME Prüfrahmen in Ihrer KNIME Analytics Platform (KAP). Das Framework ermöglicht es Ihnen, die Prüfung von Workflows und Erweiterungen zu automatisieren.

2. Erstellen Sie einen Test-Workflow, der die Funktionalität Ihrer Erweiterung validiert. Für mehr

Details, siehe Blog-Post: <http://www.knime.com/blog/enter-the-era-of-automatisierte-Arbeitsablaufprüfung-und-validierung>

3. Testen Sie Ihre Erweiterung gegen den Test-Workflow: validiert es Ihre Funktionalität und wie erwartet?

## Beitrag

Um Ihre Erweiterung auf dem KNIME Community Hub zu veröffentlichen, folgen Sie den Schritten im Leitfaden:

[Veröffentlichen Sie Ihre Erweiterung auf KNIME Community Hub](#). Diese Anleitung enthält detaillierte Anweisungen zur Einreichung Ihrer Verlängerung für Überprüfung und Veröffentlichung.

## Zurück, aufräumen

ANHANG Warten Sie auf KNIME, um Ihre Eingabe zu überprüfen und Feedback zu geben.

2. Sobald Ihre Erweiterung auf der nächsten experimentellen Community-Erweiterung Hub verfügbar ist, testen Sie es erneut mit Ihrem Test-Workflow. Die nächste experimentelle Update-Site kann

Zugang zu: <https://update.knime.com/community-contributions/trunk> (Zurzeit alle Python Erweiterungen bleiben auf dieser Seite.)

3. Laden Sie Ihren Test-Workflow auf den Community Workflow Server hoch. Dazu:

a. Öffnen Sie die KNIME Explorer-Ansicht in der KNIME Analytics Platform.

- B. Wenn Sie keinen Befestigungspunkt für den Community Workflow Server haben, klicken Sie auf den Taste in der oberen rechten Ecke der Explorer Ansicht und wählen Explorer Einstellungen konfigurieren .
- c. Erstellen Sie einen neuen Mount Point mit einer benutzerdefinierten ID und wählen Sie KNIME Community Server als Befestigungspunkttyp.
- d. Melden Sie sich an mit Ihren KNIME Forum Anmeldeinformationen (Sie müssen haben Gemeinschaft Beitragsstatus )
- e. Erstellen Sie eine neue Workflow-Gruppe innerhalb Testläufe/Trunk , geben Sie ihm einen aussagekräftigen Namen, und laden Sie Ihren Workflow(s) auf diese Gruppe.



Wenn Sie Workflows auf den Community Workflow Server hochladen, stellen Sie sicher, dass die Berechtigungen werden gesetzt, um Lesezugriff für alle zu ermöglichen. Dies stellt sicher, dass andere Benutzer können auf Ihre Workflows zugreifen und testen.

Glückwunsch! Wenn Sie Ihre Erweiterung auf KNIME Community Hub veröffentlichen, sind Sie einen Beitrag zur KNIME-Community und ermöglicht anderen, von Ihrer Arbeit zu profitieren. Danke. Sie für Ihren Beitrag!



Die nächtliche experimentelle Update-Site wird verwendet, um Erweiterungen zu testen, bevor sie werden offiziell veröffentlicht. Erweiterungen auf dieser Seite können instabil sein und sind nur für Testzwecke bestimmt.

## Anpassen der Python ausführbar

Einige Erweiterungen haben möglicherweise zusätzliche Anforderungen, die nicht Teil des gebündelten sind

Umwelt z.B. bei Drittmodellen. Für diese Erweiterungen ist es möglich

überschreiben Sie das für die Ausführung verwendete Python ausführbar. Dies kann über das System erfolgen

Eigentum `knime.python.extension.config` die auf eine spezielle YAML-Datei auf Disc zeigen muss.

Fügen Sie es zu Ihrem `knime.ini` mit der folgenden Zeile:

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



Der vordere Slash / muss auf allen Betriebssystemen verwendet werden, auch unter Windows.

Das Format des YAML ist:

```
id.of.first.extension:
conda_env_path: Pfad/zu/pixi.toml
id.of.second.extension:
python_executable: path/to/python/executable
```

Sie haben zwei Optionen, um einen benutzerdefinierten Python ausführbar anzugeben:

- Über die `conda_env_path` Das ist der Fall. Eigentum (empfohlen), das auf eine Python Umwelt auf der Maschine.
- Über die `python_executable` Eigenschaft, die auf ein ausführbares Skript verweist, das beginnt [Python \(siehe Manuell konfiguriert Python Umgebungen\)](#) Sektion in KNIME Python Integration Guide für weitere Details).

Wenn Sie beide angeben, dann `conda_env_path` Das ist der Fall. wird Vorrang haben. Es ist Ihre Verantwortung sicherzustellen, dass der Python, den Sie in dieser Datei angegeben haben, die erforderlichen Abhängigkeiten hat, um die Erweiterung. Wie oben dargestellt, können Sie das Python ausführbar von mehreren überschreiben Erweiterungen.

# Registrierung von Python Erweiterungen während Entwicklung

Um eine Python-Erweiterung zu registrieren, die Sie entwickeln, können Sie sie dem

`knime.python.extension.config`

YAML oben erläutert durch Hinzufügen einer `Src`-Eigenschaft:

```
id.of.your.dev.extension:
src: /src
conda_env_path: /.pixi/envs/default
debug_mode: true
```

Beachten Sie, dass Sie entweder

Das ist der Fall.

oder `python_executable`

weil

Analytics Platform hat keine gebündelte Umgebung für Ihre installierte Erweiterung. Für

debugging es ist auch ratsam, den Debug-Modus durch Einstellung zu aktivieren

`debug_mode: true`

. Die

debug mode deaktiviert Caching von Python-Prozessen, die einige Ihrer Code-Änderungen ermöglicht

sofort in der Analytics Plattform angezeigt werden. Diese Änderungen umfassen:

- Änderungen an der Ausführung und Konfiguration der Laufzeitlogik.
- Änderungen an bestehenden Parametern, z.B. Änderung der `EtikettenArgument`.
- Andere Änderungen, wie das Hinzufügen eines Knotens oder das Ändern einer Knotenbeschreibung, erfordern einen Neustart der Analytics-Plattform wirksam werden.
- Nicht zuletzt erfordert die vollständige Aktivierung und Deaktivierung des Debug-Modus auch einen Neustart.

## Weitere Themen

### Protokoll

Sie können die [Einloggen](#) Python-Modul, um Warnungen und Fehler an die KNIME Analytics zu senden Plattformkonsole. Durch das Gehen Datei → Vorlieben → KNIME → KNIME GUI, Sie können wählen die Console View Log Level. Jede aufeinanderfolgende Ebene umfasst die vorherigen Ebenen (d.h. DEBUG auch Nachricht von INFO, WARN, und ERROR in der Konsole durchkommen, während WARN nur erlauben WARN und ERROR Ebenen der Nachrichten).

In Ihrem Python-Skript können Sie den Logger initiieren und diese verwenden, um Nachrichten an die KNIME Analytics Plattformkonsole wie folgt:

```
# andere verschiedene Importe einschließlich Rime. Verlängerung
Importprotokollierung

LOGGER = logging.getLogger(__name__)

# your node definition via knext decorators
Klasse MyNode:
# Ihr Konfigurationsdialog Parameterdefinitionen

def konfigurieren ( ... :
    ...

    LOGGER.debug("Diese Nachricht wird in der KNIME Analytics Plattform angezeigt
    Konsole auf der DEBUG Ebene")
    LOGGER.info("Dieser wird auf der INFO-Ebene angezeigt.")
    LOGGER.warning("Dieser auf WARN-Ebene.")
    LOGGER.error("Und dies wird als ERROR-Nachricht angezeigt.")
    ...

Def ausführen( ... :
    ...

    LOGGER.info("Logger-Nachrichten können überall in Ihren Code eingefügt werden.")
    ...
```

### Gateway Cache

Um ein reibungsloses Nutzererlebnis zu ermöglichen, sperrt die Analytics Plattform die Gateways für nicht ausgeführte Aufgaben (wie z.B. die Spe-Propagation oder die Einstellungsvalidierung) der letzten verwendet Python Erweiterungen. Dieser Cache kann über zwei Systemeigenschaften konfiguriert werden:

- `knime.python.extension.gateway.cache.size` : steuert, wie viele Erweiterungen

Das Tor ist geätzt. Wenn der Cache voll ist und ein Gateway für eine neue Erweiterung angefordert wird, dann wird das Gateway der zuletzt verwendeten Erweiterung aus dem Cache gelöscht. Die Standardwert ist 3.

- `knime.python.extension.gateway.cache.expiration` : kontrolliert den Zeitraum Sekunden, nach denen ein ungenutztes Gateway aus dem Cache entfernt wird. Der Standard ist 300 Sekunden.

Die `debug_mode: true` zielgerichtet `config.yml` besprochen vor effektiv deaktiviert Cache für einzelne Erweiterungen. Standardmäßig verwenden alle Erweiterungen Caching.

# Fehlerbehebung

Falls Sie Probleme bei der Entwicklung von rein-Python-Knoten eingehen, sind hier einige nützliche Tipps, um helfen Ihnen, mehr Informationen zu sammeln und vielleicht sogar das Problem selbst zu lösen. Falls Probleme bestehen und Sie um Hilfe bitten, bitte die gesammelten Informationen einschließen.

[Haben auch einen Blick auf die \[Fehlersuche\]\(#page56\) der Python-Integrationsführung.](#page56)

## Debug Informationen finden

Ressourcenreiche Informationen helfen beim Verständnis von Problemen. Nachteilige Informationen können erhalten werden auf folgende Weise.

### Zugang zum KNIME Log

Die `wohnzimmer.de` enthält während der Ausführung von Knoten eingeloggte Informationen. Um es zu erhalten, dort sind zwei Wege:

- In der KNIME Analytics Plattform: [Blick](#) → KNIME Protokoll öffnen
- In der Datei Explorer: `/.metadata/knime/knime.log`

Nicht alle eingeloggten Informationen sind erforderlich. Bitte beschränken Sie die Informationen, die Sie der Frage. Wenn die Log-Datei keine ausreichenden Informationen enthält, können Sie die Protokollierung ändern

Verbenheit in `Datei` → Vorlieben → KNIME . Sie können sogar die Informationen an der Konsole anmelden

im KNIME Analytics Programm: `Datei` → Vorlieben → KNIME → KNIME GUI .

## Informationen 1/4ber die Python-Umgebung

wenn Conda wird verwendet, um die Informationen über die verwendete Python-Umgebung zu erhalten über:

ANH  
ANG conda aktivieren

2. conda env export

wenn Pixi wird verwendet, erhalten Sie die Informationen über die verwendete Python-Umgebung finden Sie in

die `Pixi.toml` und `Pixi.lock` Dateien in der Wurzel des Erweiterungsordners (siehe Schritt 5 der [Tutorial](#page6)) Die `Pixi.lock` Datei enthält die Informationen über die Python-Version und die verwendete

Pakete. Alternativ können Sie auch die folgenden Befehle ausführen:

- ANH  
ANG

Pixi info

um allgemeine Informationen zu erhalten
2. Pixi-Liste

die Liste der installierten Pakete erhalten
3. Pixi Schloss

um sicherzustellen, dass 

Pixi.lock

 Datei ist aktuell

Wie man Python-Version aktualisiert

[<a href="#page6" style="color: #ff6600; text-decoration: underline;">Schritt 6: Python-Umgebung erstellen, die Sie für Ihre Erweiterung verwenden.](#page6)

Für die Installation werden drei Module angegeben:

- ANH  
ANG

knime-extension

bringt alle notwendigen API-Dateien, so dass Sie Code verwenden können-  
  
Abschluss in Ihrem Editor, wenn die Umgebung dort aktiviert ist.
2. knime-python-Basis

ist ein Metapaket, das Abhängigkeiten wie Pyarrow und  
pandas etc., die zur Interaktion mit der KNIME Analytics Platform notwendig sind. wenn  
  
ihr seht euch an [die Dateien auf Anaconda.org](#) Sie sehen, dass wir 

knime-python-Basis

 bis zu  
Python 3.11.
3. Python lässt Sie die Version angeben. wie Sie in 2. sehen können, die Versionsbereich gemacht  
erhältlich von 

knime-python-Basis

 3.8-3.11.

Sie können eine Umgebung mit einer neueren Python-Version wie folgt erstellen:

conda erstellen - Nein. python = 3.11 knime-python-base knime-extension -c  
knime -c conda-forge

oder einfach durch Aktualisierung 

Pixi.toml

 Datei, die eine Zeile wie enthalten sollte:

(abhängigkeiten)  
python = ""  
knime-python-base = ""

Dann lauf 

Pixi Schloss

 zum Update der 

Pixi.lock

 Datei. Dies wird die Python-Version in der  
Umgebung zur gewünschten Version.

Entwickeln Sie mehrere Erweiterungen auf einmal

Wenn Sie mehrere Erweiterungen gleichzeitig in Ihrem KNIME Analytics entwickeln und testen möchten  
Plattform, Sie können die 

config.yml

 (siehe Schritt 5 der [Schritt 6: Python-Umgebung erstellen, die Sie für Ihre Erweiterung verwenden.](#page6)) die notwendigen  
Informationen zu weiteren Erweiterungen wie dieser:

```
src:
conda_env_path:
debug_mode: true
```

```
src:
conda_env_path:
debug_mode: true
```



Die Einbuchtung ist notwendig und muss in jeder eingezeichneten Zeile gleich sein, z.B. 2 oder 4 Räume.

## Fehler während der Last

Wenn Sie während der Entwicklung einen Fehler ähnlich erhalten

```
Loading model settings versagt: Parameter fehlt für key
```

dann ist dies wahrscheinlich, weil Sie den Parameter neu eingeführt. Neuausführung des Knotens sollte das lösen. Alternativ ziehen und fallen Sie den Knoten wieder aus dem Knoten-Repository.



Während der Development müssen Sie die Knoten immer in Ihre ziehen und fallen Workflow, wenn Sie etwas außerhalb der Methode.

Ausführung oder Konfiguration

## Spalte ist von Typ lang, aber Int wurde gesucht

Durch Inkonsistenzen der verschiedenen Betriebssysteme, ganze Spalten im Ausgangstisch kann vom Typ lang sein. Um das zu verhindern, folgen Sie diesem Beispiel:

```
def ausführen(self, exec_context, input_table):
    Import numpy als np
    df = input_table.to_pandas()
    # Nehmen wir an, df hat eine Spalte 'column1 '
    df['column1'] = df['column1'].astype(np.int32)
    Zurück knext. Tabelle.from_pandas(df)
```

## LZ4/jnijavacpp.dll/Columnar Tabelle Backend Fehler

Unter Windows können die folgenden zwei Fehler auftreten, wenn Sie zwei KNIME Analytics Platform haben Versionen geöffnet und beide verwenden den Columnar Tisch Backend. Schließen Sie beide und starten Sie nur einen.

```
ArrowColumnStoreFactory Versäumt, LZ4 Bibliotheken zu initialisieren. Der Columnar Tisch
Backend funktioniert nicht richtig.
java.lang.UnsatisfiedLinkFehler: java.io.FileNotFoundException:
C:\...\javacpp\cache\windows-x86_64\jniavacpp.dll (Der Prozess kann nicht auf die Datei zugreifen
weil es von einem anderen Verfahren verwendet wird)
```

```
ERROR : KNIME-Worker-3-Data Generator 3:18 : Node : Datengenerator : 3:18 : Ausführung
gescheitert: Unfähig, DataContainerDelegate für ColumnarTableBackend zu erstellen.
java.lang.IllegalStateException: Unable to create DataContainerDelegate for
ColumnarTableBackend.
bei
org.knime.core.data.columnar.ColumnarTableBackend.create(ColumnarTableBackend.java:115)
...
...
...
Von: java.lang.UnsatisfiedLinkFehler: java.io.FileNotFoundException:
C:\...\javacpp\cache\windows-x86_64\jniavacpp.dll (Der Prozess kann nicht auf die Datei zugreifen
weil es von einem anderen Verfahren verwendet wird)
```

## Nicht erstellen Instanz Fehler

Der folgende Fehler kann auftreten, wenn die Erweiterung mit `build_python_extension` für neuere KNIME Analytics Platform (KAP) Version. Laufen `build_python_extension` Script mit der Parameter für die spezifische KAP-Version oder eine ältere KAP-Version, z. `build_python_extension.py --knime-version 5.7`.

```
ERROR CoreUs konnte keine Instanz von Knoten erstellen
org.knime.python3.nodes.extension.ExtensionNodeSetFactory$DynamicExtensionNodeFactory:
Kann nicht initialisieren Klasse org.knime.python3.nodes.CloseablePythonNodeProxy
```

## SSL-Fehler bei der Ausführung

Wenn Sie während der Ausführung eines Knotens einen SSL-Fehler erkennen, könnte dies auf die Verwendung eines Selbstsignierte Bescheinigung. Wenn andere Knoten wie der GET Request-Knoten funktionieren, aber der Python node nicht, Sie können die Python-Knoten konfigurieren, um die gleichen Zertifikate wie die KNIME Analytics Platform. Um dies zu tun, fügen Sie die folgende Zeile zu Ihrem `knime.ini` Datei:

```
-Dknime.python.cacerts=AP
```

Dies wird die `CA_CERTS` und `ANFORDERUNGEN` Umgebungsvariablen zu einem neu erstelltes CA-Bündnis, das die zertifizierenden Behörden der KNIME Analytics Platform enthält Vertrauen. Die Python-Knoten vertrauen dann den gleichen Zertifikaten wie die KNIME Analytics Plattform.

## Einschränkungen der Pixi-Umgebung

Seit der KNIME Analytics Platform 5.5.0 wird Pixi zur Bündelung der benötigten Python-Umgebung verwendet.

für die Erweiterung. Pixi löst Conda und Pip Abhängigkeiten gleichzeitig und kann sein

strenger über Inkompatibilitäten zwischen conda und pip Paketen im Vergleich zu

vorheriger Ansatz mit `conda-lock`. Es wird im Allgemeinen empfohlen, Pixi für neue

Erweiterungen. bestehende Erweiterungen können nach Pixi migriert werden, indem man [Migration Tutorial](#).  
`Pixi.toml` Dateien können mit der `pixi init -i env.yml` Befehl, wo `env.yml` ist

eine bestehende Umgebungsdatei, die Sie zuvor verwendet haben. Die `Pixi.lock` Datei wird erstellt von `run`

`Pixi Schloss` und enthält die gelösten Abhängigkeiten. Es wird im Allgemeinen empfohlen, zu verwenden

conda-Pakete für die Abhängigkeiten und nur pip-Pakete verwenden, wenn sie nicht als

conda Pakete. Es sei darauf hingewiesen, dass für den Bündelungsprozess Pip-Quellenpakete sind

nicht unterstützt, so müssen Sie Pip-Räder oder (bessere) conda-Pakete für alle verwenden

Abhängigkeiten.

Wenn Ihre Erweiterung Pip-Source-Pakete verwendet, müssen Sie sie zu Pip-Rädern oder conda migrieren

Pakete und hosten sie auf einem Conda-Kanal oder PyPI. Für Pakete, die eine Zusammenstellung erfordern

aus nativem Code, diese müssen im Voraus gebaut werden, um mögliche Bauausfälle zu verhindern oder

Kompatibilitätsprobleme bei Anwendersystemen. Die Erstellung von nativem Code wird hier nicht beschrieben. wenn

es ist ein reines Python-Paket, das kann oft in einigen Schritten getan werden. Die genauen Schritte

abhängig vom Paket und seinen Abhängigkeiten. Hier geben wir ein Beispiel für die Migration

ein Pip-Source-Paket zu einem conda-Paket, mit [grauskull](#).

```
# Erstellen Sie eine Pixi-Umgebung für diesen Prozess
Pixi init
pixi add greyskull anaconda-client conda-build pip
```

```
# Quellpaket herunterladen
python -m pip download --no-binary :all: --nodeps = > version >
```

```
# Verwenden Sie Greyskull, um ein Conda-Paket aus dem Quellpaket zu erstellen
grauskull pypi .tar.gz
```

Stellen Sie sicher, dass das Paket als reines Python-Paket gebaut wird. Wenn nicht bereits hinzugefügt

greyskull, Sie können die folgenden Zeilen in die `Meta.yaml` Datei im erstellten conda Paket

Ordner:

```
bauen:
noarch: python
```

Darüber hinaus stellen Sie sicher, dass alle Namen übereinstimmen. Besonders `ja` und `ja` werden oft verwendet

austauschbar in Pip-Paketnamen. Wenn alles in Ordnung ist, können Sie die

conda Paket zu einem conda Kanal. Wenn Sie noch keinen conda-Kanal haben, können Sie einen erstellen

auf <https://anaconda.org/> oder verwenden Anaconda-Client eine zu erstellen.

```
# Erstellen Sie ein conda-Paket
conda build
```

```
# Laden Sie das Conda-Paket auf einen Conda-Kanal hoch
anaconda Upload /path/to/conda/package.tar.bz2
```

Nachdem das Paket hochgeladen ist, können Sie überprüfen, ob es auf [anaconda.org](https://anaconda.org/) verfügbar ist und es hinzufügen

Ihr `Pixi.toml` Datei als conda Abhängigkeit:

```
(abhängigkeiten)
= ">version>
```

Wenn Ihr Paket eine Abhängigkeit von einem anderen Paket ist, müssen Sie ein Mapping aus erstellen

conda to pip, so dass pip und conda Pakete wissen, dass dies das gleiche Paket wie die

Original Pip Quellpaket. Ansonsten sehen Sie zwei Versionen des gleichen Pakets in der

Pixi-Liste Ausgabe, eines als Conda-Paket (die, die Sie gerade erstellt haben) und eines als Pip

Paket (das Original-Pip-Quellenpaket). Um dies zu tun, müssen Sie eine Datei erstellen, die aufgerufen wird conda...

`pypi-map.json`, ideal in der Wurzel Ihrer Erweiterung Ordner. Die Datei sollte ein Mapping enthalten

vom conda Paketnamen zum pip Paketnamen, z.B.:

```
{
  "conda": "pip",
  "pip": "conda"
}
```

Und dann müssen Sie Pixi über diese Datei erzählen, indem Sie die folgende Zeile zu Ihrem hinzufügen

`Pixi.toml`

Datei:

(Arbeitsraum)

```
conda-pypi-map = { "" = "conda-pypi-map.json" }
```

Weitere Informationen zu Pixi und dem Mapping finden Sie in [die Pixi-Dokumentation](#) . Nach diesen Schritten sollte die Erweiterung ohne Probleme bündeln und das pip source Paket sollte durch das conda-Paket ersetzt werden.

# Installation Fehlerbehebung

Dieses Kapitel befasst sich mit gemeinsamen Installationsherausforderungen, die mit unserer Python-basierten KNIME Erweiterungen. Es bietet Lösungen und Ratschläge, um Benutzern zu helfen, diese zu verwalten und zu lösen Probleme effektiv, auf einen einfachen Setup-Prozess ausgerichtet.

## Offline Installation



Dieser gesamte Abschnitt ist nur für die KNIME Analytics Plattform relevant, die älter als 5,5.0. Neue Erweiterungen erfordern dieses Verfahren und Erweiterungen immer nicht die erforderlichen Python-Pakete enthalten.

Für die Performance bündeln wir Python-Pakete nicht mehr in Python-basierten Erweiterungen.

Wenn Sie daher Python-basierte Erweiterungen offline installieren möchten ("air-gapped")

Umwelt, bitte folgen Sie diesen Schritten zusätzlich

[Hinzufügen einer Offline-Update-Seite](#) :

ANHANG Installieren/Run a (Zeitalter) KNIME Analytics-Plattform auf einem Internetsystem

Zugang.

2. Installieren Sie alle gewünschten Erweiterungen.

3. Navigieren Sie auf die Präferenzseite "Python-based Extensions" über das Zahnrad (oder, in

der Klassiker UI: Datei → Vorlieben → KNIME → Python-basierte Erweiterungen ) und klicken

"Für die Offline-Installation erforderliche Pakete herunterladen" . Wählen Sie einen leeren Ordner in

die Pakete gespeichert werden. Nach der Auswahl eines Ordners sammelt KNIME die

benötigte Python-Pakete und laden Sie sie in den ausgewählten Ordner herunter.

L 347 vom 20.12.2013, S. 1). Nachdem der Download abgeschlossen ist, kopieren Sie diesen Ordner in das Ziel-Offline-System.

5. Auf dem Ziel-Offline-System setzen Sie die Umgebungsvariable

KNIME\_PYTHON\_PACKAGE\_REPO\_URL in den Ordner mit den heruntergeladenen Paketen.

6. Schließen Sie KNIME. Nach dem erneuten Start wird KNIME nun die bereitgestellten Pakete nutzen

für die Installation von Python-basierten Erweiterungen.



Wenn Sie sich nicht sicher sind, ob dieses Verfahren für die gewünschte Python-basierte erforderlich ist Erweiterungen versuchen nur, die Installation auf dem Ziel-Offline-System ohne die Umgebungsvariable einstellen. Die Installation wird mit einer Fehleranbindung scheitern dieser Dokumentationsabschnitt, wenn die obigen Schritte erforderlich sind. Alternativ, laufen Schritt 1-3 und überprüfen, ob Pakete heruntergeladen wurden.



Technisches Detail : Python-basierte Erweiterungen eine conda-Umgebung mit der notwendige conda und pip Pakete während der Installation. Diese Pakete sind entweder mit der Erweiterung gebündelt oder bei der Installation heruntergeladen. Wenn Erweiterung bündelt die Pakete ist es möglich, es von einem Reißverschluss-Update zu installieren-Website auf einem System, das keinen Internetzugang hat. Wenn die Erweiterung nicht bündelt die Pakete, die oben beschriebenen zusätzlichen Schritte für eine Offline benötigt werden Installation.

## Benutzerdefinierte conda-Umgebung Standort im Fall von Windows-Langweg Installationsprobleme

Python-basierte Erweiterungen installieren eine dedizierte conda-Umgebung mit dem Python

Pakete für diese Erweiterung erforderlich. Standardmäßig erstellt KNIME diese conda

Umgebungen am Standort: „/bundling/envs/“ für

Linux und Mac und „\bundling\envs\“ für Windows.

Es ist jedoch möglich, das Verzeichnis zu ändern, in dem die conda-Umgebungen von

Einstellung der Umgebungsvariable `KNIME_PYTHON_BUNDLING_PATH` in das gewünschte Verzeichnis. Das

kann nützlich sein, um Installationsprobleme aufgrund der Einschränkung von Bahnlängen in

Windows.



Conda-Umgebungen auf diesem Weg werden bei der Installation einer Erweiterung mit demselben Namen. Auch bei der Deinstallation einer Erweiterung, der conda Umwelt wird gelöscht.



Beim Ändern dieser Umgebungsvariable, zuvor installiert Erweiterungen, die Sie können sich auf eine Python-Umgebung verlassen. Es wird empfohlen, Sie nur diese Variable für neue KNIME-Installationen festlegen.

## Proxy-Ausgaben

Wenn Sie hinter einem Proxy stehen, können Sie Probleme bei der Installation von Python-basierten KNIME Erweiterungen.

Da Python-basierte KNIME-Erweiterungen über conda installiert werden, müssen Sie den Proxy einstellen

Einstellungen in der Konfiguration von conda. Die Konfigurationsdatei conda befindet sich in der `~/conda`

Datei. Weitere Informationen zu dieser Datei finden Sie im folgenden Tutorial:

[Wie zu benutzen](#)

[CondaRC](#) .

Um sicherzustellen, dass die Python-basierte Erweiterungsinstallation Ihre Proxy-Einstellungen respektiert, wir

die folgenden Zeilen in die `~/conda` Datei:

```
proxy_servers:  
http://domainname\username:password@proxyserver:port  
http://domainname\username:password@proxyserver:port
```

Diese sorgen dafür, dass die Proxy-Einstellungen für conda korrekt eingestellt werden. Proxy Einstellungen von KNIME wird leider nicht für die Installation propagiert, da dies die bestehende Proxy-Einstellungen für conda.

[ Verwenden Sie keine Tabs, muss ein Raum zwischen `http:` und `a href="http://..." class="bar"> http://...&#8203;`. Wenn Sie don't einen Benutzernamen und ein Passwort für die proxy, lassen Sie den `username:password` aus.

Wenn Sie in den folgenden Fehler laufen:

```
kritische libmamba Fehler herunterladen (35) SSL-Verbindungsfehler  
[https://conda.anaconda.org/conda-forge/noarch/  
schannel: next InitializeSecurityContext versagt: Unbekannter Fehler (0x80092012) - Der  
Eine Widerrufsfunktion konnte den Widerruf der Bescheinigung nicht überprüfen.
```

versuchen, `ssl_no_revoke: wahr` zu dir `~/.condarc` Datei. Der Fehler bedeutet, dass Ihr Proxy ist nicht konfiguriert, um SSL-Zertifikat-Verzichtskontrollen weiterzuleiten.

Wenn keine dieser Tipps funktionieren, können Sie auch eine Offline-Installation der Erweiterung als beschrieben in: [Offline Installation](#) .

KNIME AG  
Talacker 50  
8001 Zürich, Schweiz  
[www.knime.com](http://www.knime.com)  
[Info@knime.com](mailto:Info@knime.com)