Join us for Beam Summit 2025 this July 8th & 9th

in New York City and get the chance to attend the Google Cloud Dataflow training

LANGUAGES

Java SDK overview

Java SDK API reference 🔀

Java SDK dependencies

Java 3rd party extensions

Nexmark benchmark suite

TPC-DS benchmark suite

Python SDK overview

quickstart

Python

Java multi-language pipelines

Python SDK API reference

Python SDK dependencies

Python streaming pipelines

Ensuring Python type safety

Python multi-language pipelines

Python multi-language pipelines

Python Unrecoverable Errors

Python SDK image build

Go SDK API reference

Typescript SDK overview

Scio SDK API reference

Yaml User Defined Functions

Yaml overview

Yaml Aggregation

Error handling

Yaml Testing

Inlining Python

Yaml Providers

YAML API reference

YAML Examples

Apache Calcite dialect

Beam SQL extensions

Differences from pandas

DataFrame API reference

Example pipelines 🔀

Yaml Join

Overview

Shell

DataFrames

Overview

Walkthrough

ZetaSQL dialect

SQL

Typescript SDK API reference

Go SDK dependencies

Cross compilation

Typescript

Scala

Yaml

Go SDK overview

Machine Learning

Managing pipeline

dependencies

quickstart

Java SDK extensions

Java

Using PyTransform from YAML

Beam YAML provides the ability to easily invoke Python transforms via the PyTransform type, simply referencing them by fully qualified name. For example,

```
- type: PyTransform
config:
constructor: apache_beam.pkg.module.SomeTransform
args: [1, 'foo']
kwargs:
baz: 3
```

that returns a PTransform. Note, however, that PTransforms that do not accept or return schema'd data may not be as useable to use from YAML. Restoring the schema-ness after a non-schema returning transform can be done by using the callable option on MapToFields which takes the entire element as an input, e.g.

will invoke the transform apache_beam.pkg.mod.SomeTransform(1, 'foo', baz=3). This fully qualified name can be any PTransform class or other callable

```
- type: PyTransform
 config:
   constructor: apache_beam.pkg.module.SomeTransform
   args: [1, 'foo']
    kwargs:
       baz: 3

    type: MapToFields

 config:
   language: python
   fields:
      col1:
        callable: 'lambda element: element.col1'
        output_type: string
      col2:
        callable: 'lambda element: element.col2'
        output_type: integer
```

This can be used to call arbitrary transforms in the Beam SDK, e.g.

```
pipeline:
  transforms:
    - type: PyTransform
      name: ReadFromTsv
      input: {}
      config:
        constructor: apache_beam.io.ReadFromCsv
        kwargs:
          path: '/path/to/*.tsv'
           sep: '\t'
           skip_blank_lines: True
           true_values: ['yes']
           false_values: ['no']
           comment: '#'
           on_bad_lines: 'skip'
           binary: False
           splittable: False
```

Defining a transform inline using ___constructor___

If the desired transform does not exist, one can define it inline as well. This is done with the special <u>constructor</u> keywords, similar to how cross-language transforms are done.

With the __constuctor__ keyword, one defines a Python callable that, on invocation, returns the desired transform. The first argument (or source keyword argument, if there are no positional arguments) is interpreted as the Python code. For example

```
- type: PyTransform
  config:
    constructor: __constructor__
    kwargs:
    source: |
        def create_my_transform(inc):
        return beam.Map(lambda x: beam.Row(a=x.col2 + inc))
    inc: 10
```

will apply beam.Map(lambda x: beam.Row(a=x.col2 + 10)) to the incoming PCollection.

As a class object can be invoked as its own constructor, this allows one to define a beam.PTransform inline, e.g.

```
- type: PyTransform
config:
    constructor: __constructor__
    kwargs:
    source: |
        class MyPTransform(beam.PTransform):
        def __init__(self, inc):
            self._inc = inc
        def expand(self, pcoll):
            return pcoll | beam.Map(lambda x: beam.Row(a=x.col2 + self._inc))

inc: 10
```

which works exactly as one would expect.

Defining a transform inline using __callable__

The <u>__callable__</u> keyword works similarly, but instead of defining a callable that returns an applicable <u>PTransform</u> one simply defines the expansion to be performed as a callable. This is analogous to BeamPython's <u>ptransform.ptransform_fn</u> decorator.

In this case one can simply write

```
- type: PyTransform
config:
    constructor: __callable__
    kwargs:
    source: |
        def my_ptransform(pcoll, inc):
        return pcoll | beam.Map(lambda x: beam.Row(a=x.col2 + inc))
    inc: 10
```

External transforms

One can also invoke PTransforms define elsewhere via a python provider, for example

These can be defined inline as well, with or without dependencies, e.g.

```
pipeline:
 transforms:
   - type: ToCase
     input: ...
     config:
       upper: True
providers:
 - type: python
   config: {}
   transforms:
      'ToCase': |
       @beam.ptransform_fn
       def ToCase(pcoll, upper):
         if upper:
            return pcoll | beam.Map(lambda x: str(x).upper())
           return pcoll | beam.Map(lambda x: str(x).lower())
```





Start Overview

Quickstart (Java)

Quickstart (Go)

Quickstart

Downloads

(Python)

Concepts Pipelines Runners

Docs

Community Resources Contribute Media

Events/Meetups

Contact Us

Blog GitHub







© The Apache Software Foundation | Privacy Policy | RSS Feed

Apache Beam, Apache, Beam, the Beam logo, and the Apache feather logo are either registered trademarks or trademarks of The Apache Software Foundation. All other products or name brands are trademarks of their respective holders, including The Apache Software Foundation.