Beam Basics

November 22, 2022

Kata1.: Your first kata is to create a simple pipeline that takes a hardcoded input element "Hello Beam".

[4]:
```
import apache_beam as beam


with beam.Pipeline() as p:
    (p | beam.Create(['Hello Beam'])
       | beam.Map(print))
```

```
WARNING:apache_beam.runners.interactive.interactive_environment:Dependencies
required for Interactive Beam PCollection visualization are not available,
please use: `pip install apache-beam[interactive]` to install necessary
dependencies to enable all data visualization features.
```

```
Hello Beam
```

ParDo is a Beam transform for generic parallel processing. The ParDo processing paradigm is similar to the "Map" phase of a Map/Shuffle/Reduce-style algorithm: a ParDo transform considers each element in the input PCollection, performs some processing function (your user code) on that element, and emits zero, one, or multiple elements to an output PCollection

ParDo Kata2: Please write a simple ParDo that maps the input element by multiplying it by 10.

[6]:
```
class MultiplyByTenDoFn(beam.DoFn):

    def process(self,element):
        return [10*element]



with beam.Pipeline() as p:
    (p | beam.Create([1, 2, 3, 4, 5])
       | beam.ParDo(MultiplyByTenDoFn())
       | beam.Map(print))
```

```
10
20
30
```

```
40
50
```

ParDo OneToMany

Kata3: Please write a ParDo that maps each input sentence into words tokenized by whitespace (" ").

[7]:
```
class BreakIntoWordsDoFn(beam.DoFn):
    def process(self,element):
        return element.split(" ")




with beam.Pipeline() as p:
    (p | beam.Create(['Hello Beam', 'It is awesome'])
       | beam.ParDo(BreakIntoWordsDoFn())
       | beam.Map(print))
```

```
Hello
Beam
It
is
awesome
```

MapElements The Beam SDKs provide language-specific ways to simplify how you provide your DoFn implementation.

Kata4: Implement a simple map function that multiplies all input elements by 5 using Map

[9]:
```
with beam.Pipeline() as p:
    (p | beam.Create([10, 20, 30, 40, 50])
     | beam.Map(lambda x: 5*x)
     |  beam.Map(print))
```

```
50
100
150
200
250
```

FlatMapElements The Beam SDKs provide language-specific ways to simplify how you provide your DoFn implementation.

FlatMap can be used to simplify DoFn that maps an element to multiple elements (one to many).

Kata5: Implement a function that maps each input sentence into words tokenized by whitespace (" ") using FlatMap

[10]:
```
with beam.Pipeline() as p:
    (p | beam.Create(['Apache Beam', 'Unified Batch and Streaming'])
     | beam.FlatMap(lambda x: x.split(" "))
     | beam.Map(print))
```

```
Apache
Beam
Unified
Batch
and
Streaming
```

GroupByKey GroupByKey is a Beam transform for processing collections of key/value pairs. It's a parallel reduction operation, analogous to the Shuffle phase of a Map/Shuffle/Reduce-style algorithm. The input to GroupByKey is a collection of key/value pairs that represents a multimap, where the collection contains multiple pairs that have the same key, but different values. Given such a collection, you use GroupByKey to collect all of the values associated with each unique key.

Kata6: Implement a GroupByKey transform that groups words by its first letter.

[11]:
```
with beam.Pipeline() as p:
    (p | beam.Create(['apple', 'ball', 'car', 'bear', 'cheetah', 'ant'])
     | beam.GroupBy(lambda x:x[0])
     | beam.Map(print))
```

```
('a', ['apple', 'ant'])
('b', ['ball', 'bear'])
('c', ['car', 'cheetah'])
```

Combine - Simple Function Combine is a Beam transform for combining collections of elements or values in your data. When you apply a Combine transform, you must provide the function that contains the logic for combining the elements or values. The combining function should be commutative and associative, as the function is not necessarily invoked exactly once on all values with a given key. Because the input data (including the value collection) may be distributed across multiple workers, the combining function might be called multiple times to perform partial combining on subsets of the value collection.

Simple combine operations, such as sums, can usually be implemented as a simple function.

Kata7: Implement the summation of numbers using CombineGlobally.

[12]:
```
def sum(numbers):
    j=0
    for i in numbers:
        j=j+i

    return j

with beam.Pipeline() as p:
    (p | beam.Create([1, 2, 3, 4, 5])
```

```
    | beam.CombineGlobally(sum)
    | beam.Map(print))
```

```
15
```

Combine - Combine PerKey After creating a keyed PCollection (for example, by using a GroupByKey transform), a common pattern is to combine the collection of values associated with each key into a single, merged value. This pattern of a GroupByKey followed by merging the collection of values is equivalent to Combine PerKey transform. The combine function you supply to Combine PerKey must be an associative reduction function or a subclass of CombineFn.

Kata8: Implement the sum of scores per player using CombinePerKey.

[14]:
```
PLAYER_1 = 'Player 1'
PLAYER_2 = 'Player 2'
PLAYER_3 = 'Player 3'

with beam.Pipeline() as p:
    (p | beam.Create([(PLAYER_1, 15), (PLAYER_2, 10), (PLAYER_1, 100),
                      (PLAYER_3, 25), (PLAYER_2, 75)])
     | beam.CombinePerKey(sum)
     |  beam.Map(print))
```

```
('Player 1', 115)
('Player 2', 85)
('Player 3', 25)
```

Flatten Flatten is a Beam transform for PCollection objects that store the same data type. Flatten merges multiple PCollection objects into a single logical PCollection.

Kata9: Implement a Flatten transform that merges two PCollection of words into a single PCollection.

[26]:
```
with beam.Pipeline() as p:
    wordsStartingWithA =   p | 'Words starting with A' >> beam.Create(['apple',␣
 ↪'ant', 'arrow'])
    wordsStartingWithB =   p | 'Words starting with B' >> beam.Create(['ball',␣
 ↪'book', 'bow'])
    ((wordsStartingWithA,wordsStartingWithB)
    | beam.Flatten()
    | beam.Map(print))
```

```
apple
ant
arrow
ball
book
bow
```

Branching You can use the same PCollection as input for multiple transforms without consuming

the input or altering it.

Kata10: Branch out the numbers to two different transforms: one transform is multiplying each number by 5 and the other transform is multiplying each number by 10.

[36]:
```
class MultiplyByTenDoFn(beam.DoFn):

    def process(self,element):
        return [10*element]
class MultiplyByFiveDoFn(beam.DoFn):

    def process(self,element):
        return [5*element]

with beam.Pipeline() as p:
    numbers = p | beam.Create([1, 2, 3, 4, 5])
    mult5_results = numbers| beam.ParDo(MultiplyByFiveDoFn()
    mult10_results = numbers| beam.ParDo(MultiplyByTenDoFn())
```

```
  File "<ipython-input-36-ed8d0573f2cc>", line 13
    mult10_results = numbers| beam.ParDo(MultiplyByTenDoFn())
                   ^
SyntaxError: invalid syntax
```

Composite Transform Transforms can have a nested structure, where a complex transform performs multiple simpler transforms (such as more than one ParDo, Combine, GroupByKey, or even other composite transforms). These transforms are called composite transforms. Nesting multiple transforms inside a single composite transform can make your code more modular and easier to understand.

To create your own composite transform, create a subclass of the PTransform class and override the expand method to specify the actual processing logic. You can then use this transform just as you would a built-in transform from the Beam SDK. Within your PTransform subclass, you'll need to override the expand method. The expand method is where you add the processing logic for the PTransform. Your override of expand must accept the appropriate type of input PCollection as a parameter, and specify the output PCollection as the return value.

Kata11: Please implement a composite transform "ExtractAndMultiplyNumbers" that extracts numbers from comma separated line and then multiplies each number by 10.

[37]:
```
class ExtractAndMultiplyNumbers(beam.PTransform):
    def expand(self,pcoll):
```

```
        return pcoll | beam.FlatMap(lambda line: map(int, line.split(','))) |␣
↪beam.Map(lambda x:10*x)



with beam.Pipeline() as p:
    (p | beam.Create(['1,2,3,4,5', '6,7,8,9,10'])
     | ExtractAndMultiplyNumbers()
     | beam.Map(print))
```

```
10
20
30
40
50
60
70
80
90
100
```

Filter using ParDo

Kata12: Implement a filter function that filters out the even numbers by using ParDo.

[60]:
```
class FilterOutEvenNumber(beam.PTransform):
    def expand(self,pcoll):
        return pcoll | beam.Filter(lambda x: x%2==0)



with beam.Pipeline() as p:
    (p | beam.Create(range(1, 11))
       | FilterOutEvenNumber()
       | beam.Map(print))
```

```
2
4
6
8
10
```

[ ]:
```
class FilterOutEvenNumber(beam.DoFn):
    def process(self,element):
        if element%2!=1:
            yield element
```

```
with beam.Pipeline() as p:
    (p | beam.Create(range(1, 11))
     | beam.ParDo(FilterOutEvenNumber())
     | beam.Map(print))
```

Filter The Beam SDKs provide language-specific ways to simplify how you provide your DoFn implementation.

Kata13: Implement a filter function that filters out the odd numbers by using Filter.

[42]:
```
with beam.Pipeline() as p:
    (p | beam.Create(range(1, 11))
     | beam.Filter(lambda x: x%2!=0)
     | beam.Map(print))
```

```
1
3
5
7
9
```

Aggregation - Count

Kata14: Count the number of elements from an input.

[46]:
```
with beam.Pipeline() as p:
    (p | beam.Create(range(1, 11))
     | beam.combiners.Count.Globally()
     | beam.Map(print))
```

```
10
```

Aggregation - Sum

Kata15 : Compute the sum of all elements from an input.

Aggregation - Sum

Kata15: Compute the sum of all elements from an input.

[48]:
```
with beam.Pipeline() as p:
    (p | beam.Create(range(1, 11))
     | beam.CombineGlobally(sum)
     | beam.Map(print))
```

```
55
```

Aggregation - Mean

Kata16: Compute the mean/average of all elements from an input.

[50]:
```
with beam.Pipeline() as p:
    (p | beam.Create(range(1, 11))
     | beam.combiners.Mean.Globally()
     | beam.Map(print))
```

```
5.5
```

WithKeys

Kata17: Convert each fruit name into a key/value pair of its first letter and itself, e.g. apple => (‘a’, ‘apple’)

[53]:
```
def myfunc(text):
    return (text[0],text)
with beam.Pipeline() as p:
    (p | beam.Create(['apple', 'banana', 'cherry', 'durian', 'guava', 'melon'])
     | beam.Map(myfunc)
     | beam.Map(print))
```

```
('a', 'apple')
('b', 'banana')
('c', 'cherry')
('d', 'durian')
('g', 'guava')
('m', 'melon')
```