Kafka Tutorial in Python

In the following tutorial, we will discuss Apache Kafka along with its use in the Python programming language.

How to access Kafka in Python?

There are various libraries available in the Python programming language to use Kafka. Some of these libraries are described below:

S. No.	Library	Description
1	**Kafka-Python**	This is an open-source library designed by the Python community.
2	**PyKafka**	This library is maintained by Parsly and it has claimed to be a Pythonic API. However, we cannot create dynamic topics in this library like Kafka-Python.
3	**Confluent Python Kafka**	This library is provided by Confluent as a thin wrapper around librdkafka. Thus, it performs better than the above two.

Installing the Dependencies

We will use Kafka-Python for this project. So, we can install it manually using the pip installer as shown below:

Syntax:

```
$ pip install kafka-python
```

Now, let us start building the project.

Project Code

In the following example, we will create a producer that produces numbers ranging from 1 to 500 and send them to the Kafka broker. Later a consumer will read that data from the broker and keep them in a MongoDB collection.

One of the benefits of utilizing Kafka is that in case a consumer breaks down, another or fixed consumer will continue reading where the earlier one left. This is a good method to confirm that all the data is fed into the database without missing data or duplicates.

In the following example, let us create a new Python program file named produce.py and begin with importing some required libraries and modules.

File: produce.py

```
# importing the required libraries
from time import sleep
from json import dumps
from kafka import KafkaProducer
```

Explanation:

In the above snippet of code, we have imported the required libraries and modules. Now, let us initialize a new Kafka producer. Note the following parameters:

1. **bootstrap_servers = ['localhost: 9092']:** This parameter sets the host and port to contact the producer to bootstrap initial cluster metadata. It is not mandatory to set this here, as the default host and port is localhost: 9092.
2. **value_serializer = lambda x: dumps(x).encode('utf-8'):** This parameter functions on the serialization of the data before sending it to the broker. Here, we transform the data into a JSON file and encode it to UTF-8.

Let us consider the following snippet of code for the same.

File: produce.py

```
# initializing the Kafka producer
my_producer = KafkaProducer(
    bootstrap_servers = ['localhost:9092'],
    value_serializer = lambda x:dumps(x).encode('utf-8')
    )
```

Explanation:

In the above snippet of code, we have initialized the Kafka producer using the **KafkaProducer()** function, where we have used the parameters described above.

Now, we have to generate numbers ranging from 1 to 500. We can perform this using a **for**-loop where we use every number as a value in a dictionary with one key: num.

This key is used as a key of the data only, not as the key of the topic. Within the same loop, we will also send the data to a broker.

We can perform this by calling the send method on the producer and detailing the topic and the data.

Note: The value serializer will automatically transform and encode the data.

We can take five seconds of a break in order to conclude the iteration. In case we have to confirm whether the broker received the message, it is advised to include a callback.

File: produce.py

```
# generating the numbers ranging from 1 to 500
for n in range(500):
   my_data = {'num' : n}
   my_producer.send('testnum', value = my_data)
   sleep(5)
```

Explanation:

In the above snippet of code, we have used the **for**-loop to iterate the number ranging from one to 500. We have also added the interval of five seconds between each iteration.

If somebody wants to test the code, it is recommended to create a new topic and send the data to that newly generated topic. This method will avoid any case of duplicate values and possible confusion in the **testnum** topic when we will be testing the producer and consumer together.

Consuming the Data

Before we get started with the coding part of the consumer, let us create a new Python program file and name it consume.py. We will import some of the modules such as **json.loads, MongoClient** and **KafkaConsumer**. Since **PyMongo** is out of the scope of this tutorial, we won't be digging any deeper into its code.

Moreover, somebody can also replace the mongo code with any other code as per needs. We can code this in order to enter the data into another database, code to process the data, or anything else one can think of.

Let us consider the following snippet of code, to begin with.

File: consume.py

```
# importing the required modules
from json import loads
from kafka import KafkaConsumer
from pymongo import MongoClient
```

Explanation:

In the above snippet of code, we have imported the required modules from their respective libraries.

Let us create the Kafka Consumer. We will use the **KafkaConsumer()** function for this work; so let's have a closer look at the parameters used in this function.

1. **Topic:** The first parameter of the **KafkaConsumer()** function is the topic. In the following case, it is **testnum**.
2. **bootstrap_servers = ['localhost: 9092']:** This parameter is same as the producer.
3. **auto_offset_reset = 'earliest':** This parameter is among the other significant parameters. It handles where the consumer restarts reading after being turned off or breaking down and we can set it either to latest or earliest. Whenever we set it to earliest, the consumer begins reading at the latest committed offset. Whenever we set it to the latest, the consumer begins reading at the log's end. And that is exactly what we need here.
4. **enable_auto_commit = True:** This parameter confirms whether the consumer commits its read offset each interval.
5. **auto_commit_interval_ms = 1000ms:** This parameter is used to set the interval between two commits. As messages are coming in every interval of five seconds, committing every second appears to be fair.
6. **group_id = 'counters':** This parameter is the group of consumers to which the consumer belongs. Note that a consumer must be part of a consumer group in order to make them work automatically committed.
7. The value **deserializer** is used to deserialize the data into a general JSON format, the inverse of the working of the value serializer.

Let us consider the following snippet of code for the same.

File: consume.py

```
# generating the Kafka Consumer
my_consumer = KafkaConsumer(
   'testnum',
    bootstrap_servers = ['localhost : 9092'],
    auto_offset_reset = 'earliest',
    enable_auto_commit = True,
    group_id = 'my-group',
    value_deserializer = lambda x : loads(x.decode('utf-8'))
    )
```

Explanation:

In the above snippet of code, we have used the **KafkaConsumer()** function to generate the Kafka Consumer. We have also added the parameters within the function that we studied earlier.

Now, let us consider the following snippet of code to connect to the **testnum** collection (This collection is similar to a table in a relational database) of the MongoDB database.

File: consume.py

```
my_client = MongoClient('localhost : 27017')
my_collection = my_client.testnum.testnum
```

Explanation:

In the above snippet of code, we have defined a variable as **my_client** that uses the **MongoClient()** function specified with the host and port. We have then defined another variable as **my_collection** that uses the **my_client** variable to access the data in the **testnum** topic.

This data can be extracted from the consumer by looping through it (here, the consumer can be considered as an iterable). The consumer will keep listening until the broker does not respond anymore. We can access the message value using the value attribute. Here, we overwrite the message with the message value.

The next line inserts the data into the database collection. The last line will print a confirmation that the message was added to our collection.

Note: It is possible to insert callbacks to all the actions in this loop.

File: consume.py

```
for message in my_consumer:
    message = message.value
    collection.insert_one(message)
    print(message + " added to " + my_collection)
```

Explanation:

In the above snippet of code, we have used the **for**-loop to iterate through the consumer in order to extract the data. Now in order to test the code, one can execute the **produce.py** file first and then **consume.py**.