kafka-python Documentation

Release 0.9.4

David Arthur

Apr 21, 2017

Contents

This module provides low-level protocol support for Apache Kafka as well as high-level consumer and producer classes. Request batching is supported by the protocol as well as broker-aware request routing. Gzip and Snappy compression is also supported for message sets.

http://kafka.apache.org/

On Freenode IRC at #kafka-python, as well as #apache-kafka

For general discussion of kafka-client design and implementation (not python specific), see https://groups.google.com/forum/m/#!forum/kafka-clients

CHAPTER 1

Status

The current stable version of this package is 0.9.3 and is compatible with:

Kafka broker versions

- 0.8.2.1 [offset management currently ZK only – does not support ConsumerCoordinator offset management APIs]
- 0.8.1.1
- 0.8.1
- 0.8.0

Python versions

- 2.6 (tested on 2.6.9)
- 2.7 (tested on 2.7.9)
- 3.3 (tested on 3.3.5)
- 3.4 (tested on 3.4.2)
- pypy (tested on pypy 2.5.0 / python 2.7.8)

CHAPTER 2

License

CHAPTER 3

Contents

Install

Install with your favorite package manager

Latest Release

Pip:

```
pip install kafka-python
```

Releases are also listed at https://github.com/mumrah/kafka-python/releases

Bleeding-Edge

```
git clone https://github.com/mumrah/kafka-python
pip install ./kafka-python
```

Setuptools:

```
git clone https://github.com/mumrah/kafka-python
easy_install ./kafka-python
```

Using *setup.py* directly:

```
git clone https://github.com/mumrah/kafka-python
cd kafka-python
python setup.py install
```

Optional Snappy install

Install Development Libraries

Download and build Snappy from http://code.google.com/p/snappy/downloads/list

Ubuntu:

```
apt-get install libsnappy-dev
```

OSX:

```
brew install snappy
```

From Source:

```
wget http://snappy.googlecode.com/files/snappy-1.0.5.tar.gz
tar xzvf snappy-1.0.5.tar.gz
cd snappy-1.0.5
./configure
make
sudo make install
```

Install Python Module

Install the *python-snappy* module

```
pip install python-snappy
```

Tests

Run the unit tests

```
tox
```

Run a subset of unit tests

```
# run protocol tests only
tox -- -v test.test_protocol

# test with pypy only
tox -e pypy

# Run only 1 test, and use python 2.7
tox -e py27 -- -v --with-id --collect-only

# pick a test number from the list like #102
tox -e py27 -- -v --with-id 102
```

Run the integration tests

The integration tests will actually start up real local Zookeeper instance and Kafka brokers, and send messages in using the client.

First, get the kafka binaries for integration testing:

```
./build_integration.sh
```

By default, the build_integration.sh script will download binary distributions for all supported kafka versions. To test against the latest source build, set KAFKA_VERSION=trunk and optionally set SCALA_VERSION (defaults to 2.8.0, but 2.10.1 is recommended)

```
SCALA_VERSION=2.10.1 KAFKA_VERSION=trunk ./build_integration.sh
```

Then run the tests against supported Kafka versions, simply set the *KAFKA_VERSION* env variable to the server build you want to use for testing:

```
KAFKA_VERSION=0.8.0 tox
KAFKA_VERSION=0.8.1 tox
KAFKA_VERSION=0.8.1.1 tox
KAFKA_VERSION=trunk tox
```

Usage

SimpleProducer

```
from kafka import SimpleProducer, KafkaClient

# To send messages synchronously
kafka = KafkaClient('localhost:9092')
producer = SimpleProducer(kafka)

# Note that the application is responsible for encoding messages to type bytes
producer.send_messages(b'my-topic', b'some message')
producer.send_messages(b'my-topic', b'this method', b'is variadic')

# Send unicode message
producer.send_messages(b'my-topic', u'?'.encode('utf-8'))
```

Asynchronous Mode

```
# To send messages asynchronously
producer = SimpleProducer(kafka, async=True)
producer.send_messages(b'my-topic', b'async message')

# To wait for acknowledgements
# ACK_AFTER_LOCAL_WRITE : server will wait till the data is written to
#                         a local log before sending response
# ACK_AFTER_CLUSTER_COMMIT : server will block until the message is committed
#                            by all in sync replicas before sending a response
producer = SimpleProducer(kafka, async=False,
                          req_acks=SimpleProducer.ACK_AFTER_LOCAL_WRITE,
```

```
                          ack_timeout=2000,
                          sync_fail_on_error=False)

responses = producer.send_messages(b'my-topic', b'another message')
for r in responses:
    logging.info(r.offset)

# To send messages in batch. You can use any of the available
# producers for doing this. The following producer will collect
# messages in batch and send them to Kafka after 20 messages are
# collected or every 60 seconds
# Notes:
# * If the producer dies before the messages are sent, there will be losses
# * Call producer.stop() to send the messages and cleanup
producer = SimpleProducer(kafka, async=True,
                          batch_send_every_n=20,
                          batch_send_every_t=60)
```

Keyed messages

```
from kafka import (
    KafkaClient, KeyedProducer,
    Murmur2Partitioner, RoundRobinPartitioner)

kafka = KafkaClient('localhost:9092')

# HashedPartitioner is default (currently uses python hash())
producer = KeyedProducer(kafka)
producer.send_messages(b'my-topic', b'key1', b'some message')
producer.send_messages(b'my-topic', b'key2', b'this methode')

# Murmur2Partitioner attempts to mirror the java client hashing
producer = KeyedProducer(kafka, partitioner=Murmur2Partitioner)

# Or just produce round-robin (or just use SimpleProducer)
producer = KeyedProducer(kafka, partitioner=RoundRobinPartitioner)
```

KafkaConsumer

```
from kafka import KafkaConsumer

# To consume messages
consumer = KafkaConsumer('my-topic',
                         group_id='my_group',
                         bootstrap_servers=['localhost:9092'])
for message in consumer:
    # message value is raw byte string -- decode if necessary!
    # e.g., for unicode: `message.value.decode('utf-8')`
    print("%s:%d:%d: key=%s value=%s" % (message.topic, message.partition,
                                         message.offset, message.key,
                                         message.value))
```

messages (m) are namedtuples with attributes:

- *m.topic*: topic name (str)
- *m.partition*: partition number (int)
- *m.offset*: message offset on topic-partition log (int)
- *m.key*: key (bytes - can be None)
- *m.value*: message (output of deserializer_class - default is raw bytes)

```
  from kafka import KafkaConsumer

  # more advanced consumer -- multiple topics w/ auto commit offset
  # management
  consumer = KafkaConsumer('topic1', 'topic2',
                           bootstrap_servers=['localhost:9092'],
                           group_id='my_consumer_group',
                           auto_commit_enable=True,
                           auto_commit_interval_ms=30 * 1000,
                           auto_offset_reset='smallest')

  # Infinite iteration
  for m in consumer:
    do_some_work(m)

    # Mark this message as fully consumed
    # so it can be included in the next commit
    #
    # **messages that are not marked w/ task_done currently do not commit!
    consumer.task_done(m)

  # If auto_commit_enable is False, remember to commit() periodically
  consumer.commit()

  # Batch process interface
  while True:
    for m in kafka.fetch_messages():
      process_message(m)
      consumer.task_done(m)


Configuration settings can be passed to constructor,
otherwise defaults will be used:
```

```
    client_id='kafka.consumer.kafka',
    group_id=None,
    fetch_message_max_bytes=1024*1024,
    fetch_min_bytes=1,
    fetch_wait_max_ms=100,
    refresh_leader_backoff_ms=200,
    bootstrap_servers=[],
    socket_timeout_ms=30*1000,
    auto_offset_reset='largest',
    deserializer_class=lambda msg: msg,
    auto_commit_enable=False,
    auto_commit_interval_ms=60 * 1000,
    consumer_timeout_ms=-1

Configuration parameters are described in more detail at
http://kafka.apache.org/documentation.html#highlevelconsumerapi
```

Multiprocess consumer

```
from kafka import KafkaClient, MultiProcessConsumer

kafka = KafkaClient('localhost:9092')

# This will split the number of partitions among two processes
consumer = MultiProcessConsumer(kafka, b'my-group', b'my-topic', num_procs=2)

# This will spawn processes such that each handles 2 partitions max
consumer = MultiProcessConsumer(kafka, b'my-group', b'my-topic',
                                partitions_per_proc=2)

for message in consumer:
    print(message)

for message in consumer.get_messages(count=5, block=True, timeout=4):
    print(message)
```

Low level

```
from kafka import KafkaClient, create_message
from kafka.protocol import KafkaProtocol
from kafka.common import ProduceRequest

kafka = KafkaClient('localhost:9092')

req = ProduceRequest(topic=b'my-topic', partition=1,
    messages=[create_message(b'some message')])
resps = kafka.send_produce_request(payloads=[req], fail_on_error=True)
kafka.close()

resps[0].topic      # b'my-topic'
resps[0].partition  # 1
resps[0].error      # 0 (hopefully)
resps[0].offset     # offset of the first message sent in this request
```

kafka

kafka package

Subpackages

kafka.consumer package

Submodules

kafka.consumer.base module

class kafka.consumer.base.**Consumer**(*client*, *group*, *topic*, *partitions=None*, *auto_commit=True*, *auto_commit_every_n=100*, *auto_commit_every_t=5000*)

Bases: object

Base class to be used by other consumers. Not to be used directly

This base class provides logic for

- initialization and fetching metadata of partitions
- Auto-commit logic
- APIs for fetching pending message count

commit(*partitions=None*)

Commit stored offsets to Kafka via OffsetCommitRequest (v0)

Keyword Arguments **partitions** (*list*) – list of partitions to commit, default is to commit all of them

Returns: True on success, False on failure

fetch_last_known_offsets(*partitions=None*)

pending(*partitions=None*)

Gets the pending message count

Keyword Arguments **partitions** (*list*) – list of partitions to check for, default is to check all

stop()

kafka.consumer.kafka module

class kafka.consumer.kafka.**KafkaConsumer**(**topics*, ***configs*)

Bases: object

A simpler kafka consumer

commit()

Store consumed message offsets (marked via task_done()) to kafka cluster for this consumer_group.

Returns True on success, or False if no offsets were found for commit

---

Note: this functionality requires server version >=0.8.1.1 https://cwiki.apache.org/confluence/display/KAFKA/A+Guide+To+The+Kafka+Protocol#AGuideToTheKafkaProtocol-OffsetCommit/FetchAPI

---

configure(***configs*)

Configure the consumer instance

Configuration settings can be passed to constructor, otherwise defaults will be used:

Keyword Arguments

- **bootstrap_servers** (*list*) – List of initial broker nodes the consumer should contact to bootstrap initial cluster metadata. This does not have to be the full node list. It just needs to have at least one broker that will respond to a Metadata API Request.
- **client_id** (*str*) – a unique name for this client. Defaults to 'kafka.consumer.kafka'.

- **group_id** (*str*) – the name of the consumer group to join, Offsets are fetched / committed to this group name.
- **fetch_message_max_bytes** (*int, optional*) – Maximum bytes for each topic/partition fetch request. Defaults to 1024*1024.
- **fetch_min_bytes** (*int, optional*) – Minimum amount of data the server should return for a fetch request, otherwise wait up to fetch_wait_max_ms for more data to accumulate. Defaults to 1.
- **fetch_wait_max_ms** (*int, optional*) – Maximum time for the server to block waiting for fetch_min_bytes messages to accumulate. Defaults to 100.
- **refresh_leader_backoff_ms** (*int, optional*) – Milliseconds to backoff when refreshing metadata on errors (subject to random jitter). Defaults to 200.
- **socket_timeout_ms** (*int, optional*) – TCP socket timeout in milliseconds. Defaults to 30*1000.
- **auto_offset_reset** (*str, optional*) – A policy for resetting offsets on OffsetOutOfRange errors. 'smallest' will move to the oldest available message, 'largest' will move to the most recent. Any ofther value will raise the exception. Defaults to 'largest'.
- **deserializer_class** (*callable, optional*) – Any callable that takes a raw message value and returns a deserialized value. Defaults to

  lambda msg: msg.
- **auto_commit_enable** (*bool, optional*) – Enabling auto-commit will cause the KafkaConsumer to periodically commit offsets without an explicit call to commit(). Defaults to False.
- **auto_commit_interval_ms** (*int, optional*) – If auto_commit_enabled, the milliseconds between automatic offset commits. Defaults to 60 * 1000.
- **auto_commit_interval_messages** (*int, optional*) – If auto_commit_enabled, a number of messages consumed between automatic offset commits. Defaults to None (disabled).
- **consumer_timeout_ms** (*int, optional*) – number of millisecond to throw a timeout exception to the consumer if no message is available for consumption. Defaults to -1 (dont throw exception).

Configuration parameters are described in more detail at http://kafka.apache.org/documentation.html#highlevelconsumerapi

fetch_messages()

Sends FetchRequests for all topic/partitions set for consumption

Returns Generator that yields KafkaMessage structs after deserializing with the configured *deserializer_class*

Note: Refreshes metadata on errors, and resets fetch offset on OffsetOutOfRange, per the configured *auto_offset_reset* policy

---

See also:

Key KafkaConsumer configuration parameters: * *fetch_message_max_bytes* * *fetch_max_wait_ms* * *fetch_min_bytes* * *deserializer_class* * *auto_offset_reset*

get_partition_offsets(*topic*, *partition*, *request_time_ms*, *max_num_offsets*)
Request available fetch offsets for a single topic/partition

Keyword Arguments

- **topic** (*str*) – topic for offset request
- **partition** (*int*) – partition for offset request
- **request_time_ms** (*int*) – Used to ask for all messages before a certain time (ms). There are two special values. Specify -1 to receive the latest offset (i.e. the offset of the next coming message) and -2 to receive the earliest available offset. Note that because offsets are pulled in descending order, asking for the earliest offset will always return you a single element.
- **max_num_offsets** (*int*) – Maximum offsets to include in the OffsetResponse

Returns a list of offsets in the OffsetResponse submitted for the provided topic / partition. See: https://cwiki.apache.org/confluence/display/KAFKA/A+Guide+To+The+Kafka+Protocol#AGuideToTheKafkaProtocol-OffsetAPI

next()
Return the next available message

Blocks indefinitely unless consumer_timeout_ms > 0

Returns a single KafkaMessage from the message iterator

Raises ConsumerTimeout after consumer_timeout_ms and no message

---

Note: This is also the method called internally during iteration

---

offsets(*group=None*)
Get internal consumer offset values

Keyword Arguments **group** – Either "fetch", "commit", "task_done", or "highwater". If no group specified, returns all groups.

Returns A copy of internal offsets struct

set_topic_partitions(**topics*)
Set the topic/partitions to consume Optionally specify offsets to start from

Accepts types:

•str (utf-8): topic name (will consume all available partitions)

•tuple: (topic, partition)

•**dict:**

- { topic: partition }
- { topic: [partition list] }
- { topic: (partition tuple,) }

Optionally, offsets can be specified directly:

•tuple: (topic, partition, offset)

•dict: { (topic, partition): offset, ... }

Example:

```
kafka = KafkaConsumer()

# Consume topic1-all; topic2-partition2; topic3-partition0
kafka.set_topic_partitions("topic1", ("topic2", 2), {"topic3": 0})

# Consume topic1-0 starting at offset 12, and topic2-1 at offset 45
# using tuples --
kafka.set_topic_partitions(("topic1", 0, 12), ("topic2", 1, 45))

# using dict --
kafka.set_topic_partitions({ ("topic1", 0): 12, ("topic2", 1): 45 })
```

task_done(*message*)
Mark a fetched message as consumed.

Offsets for messages marked as "task_done" will be stored back to the kafka cluster for this consumer group on commit()

Parameters **message** (KafkaMessage) – the message to mark as complete

Returns True, unless the topic-partition for this message has not been configured for the consumer. In normal operation, this should not happen. But see github issue 364.

class kafka.consumer.kafka.**OffsetsStruct**(*fetch*, *highwater*, *commit*, *task_done*)
Bases: tuple

commit
Alias for field number 2

fetch
Alias for field number 0

highwater
Alias for field number 1

task_done
Alias for field number 3

kafka.consumer.multiprocess module

class kafka.consumer.multiprocess.**Events**(*start*, *pause*, *exit*)
Bases: tuple

exit
Alias for field number 2

pause
Alias for field number 1

start
Alias for field number 0

class kafka.consumer.multiprocess.**MultiProcessConsumer**(*client*, *group*, *topic*, *partitions=None*, *auto_commit=True*, *auto_commit_every_n=100*, *auto_commit_every_t=5000*, *num_procs=1*, *partitions_per_proc=0*, ***simple_consumer_options*)

Bases: *kafka.consumer.base.Consumer*

A consumer implementation that consumes partitions for a topic in parallel using multiple processes

Parameters

- **client** – a connected KafkaClient
- **group** – a name for this consumer, used for offset storage and must be unique If you are connecting to a server that does not support offset commit/fetch (any prior to 0.8.1.1), then you *must* set this to None
- **topic** – the topic to consume

Keyword Arguments

- **partitions** – An optional list of partitions to consume the data from
- **auto_commit** – default True. Whether or not to auto commit the offsets
- **auto_commit_every_n** – default 100. How many messages to consume before a commit
- **auto_commit_every_t** – default 5000. How much time (in milliseconds) to wait before commit
- **num_procs** – Number of processes to start for consuming messages. The available partitions will be divided among these processes
- **partitions_per_proc** – Number of partitions to be allocated per process (overrides num_procs)

Auto commit details: If both auto_commit_every_n and auto_commit_every_t are set, they will reset one another when one is triggered. These triggers simply call the commit method on this class. A manual call to commit will also reset these triggers

get_messages(*count=1*, *block=True*, *timeout=10*)
Fetch the specified number of messages

Keyword Arguments

- **count** – Indicates the maximum number of messages to be fetched
- **block** – If True, the API will block till some messages are fetched.
- **timeout** – If block is True, the function will block for the specified time (in seconds) until count messages is fetched. If None, it will block forever.

stop()

kafka.consumer.simple module

class kafka.consumer.simple.**FetchContext**(*consumer*, *block*, *timeout*)
Bases: object

Class for managing the state of a consumer during fetch

class kafka.consumer.simple.**SimpleConsumer**(*client*, *group*, *topic*, *auto_commit=True*, *partitions=None*, *auto_commit_every_n=100*, *auto_commit_every_t=5000*, *fetch_size_bytes=4096*, *buffer_size=4096*, *max_buffer_size=32768*, *iter_timeout=None*, *auto_offset_reset='largest'*)
Bases: *kafka.consumer.base.Consumer*

A simple consumer implementation that consumes all/specified partitions for a topic

Parameters

- **client** – a connected KafkaClient
- **group** – a name for this consumer, used for offset storage and must be unique If you are connecting to a server that does not support offset commit/fetch (any prior to 0.8.1.1), then you *must* set this to None
- **topic** – the topic to consume

Keyword Arguments

- **partitions** – An optional list of partitions to consume the data from
- **auto_commit** – default True. Whether or not to auto commit the offsets
- **auto_commit_every_n** – default 100. How many messages to consume before a commit
- **auto_commit_every_t** – default 5000. How much time (in milliseconds) to wait before commit
- **fetch_size_bytes** – number of bytes to request in a FetchRequest
- **buffer_size** – default 4K. Initial number of bytes to tell kafka we have available. This will double as needed.
- **max_buffer_size** – default 16K. Max number of bytes to tell kafka we have available. None means no limit.
- **iter_timeout** – default None. How much time (in seconds) to wait for a message in the iterator before exiting. None means no timeout, so it will wait forever.
- **auto_offset_reset** – default largest. Reset partition offsets upon OffsetOutOfRangeError. Valid values are largest and smallest. Otherwise, do not reset the offsets and raise OffsetOutOfRangeError.

Auto commit details: If both auto_commit_every_n and auto_commit_every_t are set, they will reset one another when one is triggered. These triggers simply call the commit method on this class. A manual call to commit will also reset these triggers

get_message(*block=True*, *timeout=0.1*, *get_partition_info=None*)

get_messages(*count=1*, *block=True*, *timeout=0.1*)

Fetch the specified number of messages

Keyword Arguments

- **count** – Indicates the maximum number of messages to be fetched
- **block** – If True, the API will block till some messages are fetched.
- **timeout** – If block is True, the function will block for the specified time (in seconds) until count messages is fetched. If None, it will block forever.

provide_partition_info()

Indicates that partition info must be returned by the consumer

reset_partition_offset(*partition*)

Update offsets using auto_offset_reset policy (smallest|largest)

Parameters **partition** (*int*) – the partition for which offsets should be updated

Returns: Updated offset on success, None on failure

seek(*offset*, *whence*)
Alter the current offset in the consumer, similar to fseek

Parameters

- **offset** – how much to modify the offset
- **whence** – where to modify it from
  - 0 is relative to the earliest available offset (head)
  - 1 is relative to the current offset
  - 2 is relative to the latest known offset (tail)

Module contents

class kafka.consumer.**SimpleConsumer**(*client*, *group*, *topic*, *auto_commit=True*, *partitions=None*, *auto_commit_every_n=100*, *auto_commit_every_t=5000*, *fetch_size_bytes=4096*, *buffer_size=4096*, *max_buffer_size=32768*, *iter_timeout=None*, *auto_offset_reset='largest'*)
Bases: *kafka.consumer.base.Consumer*

A simple consumer implementation that consumes all/specified partitions for a topic

Parameters

- **client** – a connected KafkaClient
- **group** – a name for this consumer, used for offset storage and must be unique If you are connecting to a server that does not support offset commit/fetch (any prior to 0.8.1.1), then you *must* set this to None
- **topic** – the topic to consume

Keyword Arguments

- **partitions** – An optional list of partitions to consume the data from
- **auto_commit** – default True. Whether or not to auto commit the offsets
- **auto_commit_every_n** – default 100. How many messages to consume before a commit
- **auto_commit_every_t** – default 5000. How much time (in milliseconds) to wait before commit
- **fetch_size_bytes** – number of bytes to request in a FetchRequest
- **buffer_size** – default 4K. Initial number of bytes to tell kafka we have available. This will double as needed.
- **max_buffer_size** – default 16K. Max number of bytes to tell kafka we have available. None means no limit.
- **iter_timeout** – default None. How much time (in seconds) to wait for a message in the iterator before exiting. None means no timeout, so it will wait forever.
- **auto_offset_reset** – default largest. Reset partition offsets upon OffsetOutOfRangeError. Valid values are largest and smallest. Otherwise, do not reset the offsets and raise OffsetOutOfRangeError.

Auto commit details: If both auto_commit_every_n and auto_commit_every_t are set, they will reset one another when one is triggered. These triggers simply call the commit method on this class. A manual call to commit will also reset these triggers

get_message(*block=True*, *timeout=0.1*, *get_partition_info=None*)

get_messages(*count=1*, *block=True*, *timeout=0.1*)
Fetch the specified number of messages

Keyword Arguments

- **count** – Indicates the maximum number of messages to be fetched
- **block** – If True, the API will block till some messages are fetched.
- **timeout** – If block is True, the function will block for the specified time (in seconds) until count messages is fetched. If None, it will block forever.

provide_partition_info()
Indicates that partition info must be returned by the consumer

reset_partition_offset(*partition*)
Update offsets using auto_offset_reset policy (smallest|largest)

Parameters **partition** (*int*) – the partition for which offsets should be updated

Returns: Updated offset on success, None on failure

seek(*offset*, *whence*)
Alter the current offset in the consumer, similar to fseek

Parameters

- **offset** – how much to modify the offset
- **whence** – where to modify it from
  - 0 is relative to the earliest available offset (head)
  - 1 is relative to the current offset
  - 2 is relative to the latest known offset (tail)

class kafka.consumer.**MultiProcessConsumer**(*client*, *group*, *topic*, *partitions=None*, *auto_commit=True*, *auto_commit_every_n=100*, *auto_commit_every_t=5000*, *num_procs=1*, *partitions_per_proc=0*, ***simple_consumer_options*)

Bases: kafka.consumer.base.Consumer

A consumer implementation that consumes partitions for a topic in parallel using multiple processes

Parameters

- **client** – a connected KafkaClient
- **group** – a name for this consumer, used for offset storage and must be unique If you are connecting to a server that does not support offset commit/fetch (any prior to 0.8.1.1), then you *must* set this to None
- **topic** – the topic to consume

Keyword Arguments

- **partitions** – An optional list of partitions to consume the data from
- **auto_commit** – default True. Whether or not to auto commit the offsets

- **auto_commit_every_n** – default 100. How many messages to consume before a commit
- **auto_commit_every_t** – default 5000. How much time (in milliseconds) to wait before commit
- **num_procs** – Number of processes to start for consuming messages. The available partitions will be divided among these processes
- **partitions_per_proc** – Number of partitions to be allocated per process (overrides num_procs)

Auto commit details: If both auto_commit_every_n and auto_commit_every_t are set, they will reset one another when one is triggered. These triggers simply call the commit method on this class. A manual call to commit will also reset these triggers

get_messages(*count=1*, *block=True*, *timeout=10*)
Fetch the specified number of messages

Keyword Arguments

- **count** – Indicates the maximum number of messages to be fetched
- **block** – If True, the API will block till some messages are fetched.
- **timeout** – If block is True, the function will block for the specified time (in seconds) until count messages is fetched. If None, it will block forever.

stop()

class kafka.consumer.**KafkaConsumer**(**topics*, ***configs*)
Bases: object

A simpler kafka consumer

commit()
Store consumed message offsets (marked via task_done()) to kafka cluster for this consumer_group.

Returns True on success, or False if no offsets were found for commit

---

Note: this functionality requires server version >=0.8.1.1 https://cwiki.apache.org/confluence/display/KAFKA/A+Guide+To+The+Kafka+Protocol#AGuideToTheKafkaProtocol-OffsetCommit/FetchAPI

---

configure(***configs*)
Configure the consumer instance

Configuration settings can be passed to constructor, otherwise defaults will be used:

Keyword Arguments

- **bootstrap_servers** (*list*) – List of initial broker nodes the consumer should contact to bootstrap initial cluster metadata. This does not have to be the full node list. It just needs to have at least one broker that will respond to a Metadata API Request.
- **client_id** (*str*) – a unique name for this client. Defaults to 'kafka.consumer.kafka'.
- **group_id** (*str*) – the name of the consumer group to join, Offsets are fetched / committed to this group name.
- **fetch_message_max_bytes** (*int, optional*) – Maximum bytes for each topic/partition fetch request. Defaults to 1024*1024.

- **fetch_min_bytes** (*int, optional*) – Minimum amount of data the server should return for a fetch request, otherwise wait up to fetch_wait_max_ms for more data to accumulate. Defaults to 1.
- **fetch_wait_max_ms** (*int, optional*) – Maximum time for the server to block waiting for fetch_min_bytes messages to accumulate. Defaults to 100.
- **refresh_leader_backoff_ms** (*int, optional*) – Milliseconds to backoff when refreshing metadata on errors (subject to random jitter). Defaults to 200.
- **socket_timeout_ms** (*int, optional*) – TCP socket timeout in milliseconds. Defaults to 30*1000.
- **auto_offset_reset** (*str, optional*) – A policy for resetting offsets on OffsetOutOfRange errors. 'smallest' will move to the oldest available message, 'largest' will move to the most recent. Any ofther value will raise the exception. Defaults to 'largest'.
- **deserializer_class** (*callable, optional*) – Any callable that takes a raw message value and returns a deserialized value. Defaults to

  lambda msg: msg.
- **auto_commit_enable** (*bool, optional*) – Enabling auto-commit will cause the KafkaConsumer to periodically commit offsets without an explicit call to commit(). Defaults to False.
- **auto_commit_interval_ms** (*int, optional*) – If auto_commit_enabled, the milliseconds between automatic offset commits. Defaults to 60 * 1000.
- **auto_commit_interval_messages** (*int, optional*) – If auto_commit_enabled, a number of messages consumed between automatic offset commits. Defaults to None (disabled).
- **consumer_timeout_ms** (*int, optional*) – number of millisecond to throw a timeout exception to the consumer if no message is available for consumption. Defaults to -1 (dont throw exception).

Configuration parameters are described in more detail at http://kafka.apache.org/documentation.html#highlevelconsumerapi

fetch_messages()

Sends FetchRequests for all topic/partitions set for consumption

Returns Generator that yields KafkaMessage structs after deserializing with the configured *deserializer_class*

Note: Refreshes metadata on errors, and resets fetch offset on OffsetOutOfRange, per the configured *auto_offset_reset* policy

---

See also:

Key KafkaConsumer configuration parameters: * *fetch_message_max_bytes* * *fetch_max_wait_ms* * *fetch_min_bytes* * *deserializer_class* * *auto_offset_reset*

get_partition_offsets(*topic*, *partition*, *request_time_ms*, *max_num_offsets*)

Request available fetch offsets for a single topic/partition

Keyword Arguments

- **topic** (*str*) – topic for offset request
- **partition** (*int*) – partition for offset request

- **request_time_ms** (*int*) – Used to ask for all messages before a certain time (ms). There are two special values. Specify -1 to receive the latest offset (i.e. the offset of the next coming message) and -2 to receive the earliest available offset. Note that because offsets are pulled in descending order, asking for the earliest offset will always return you a single element.
- **max_num_offsets** (*int*) – Maximum offsets to include in the OffsetResponse

Returns a list of offsets in the OffsetResponse submitted for the provided topic / partition. See: https://cwiki.apache.org/confluence/display/KAFKA/A+Guide+To+The+Kafka+Protocol#AGuideToTheKafkaProtocol-OffsetAPI

next()

Return the next available message

Blocks indefinitely unless consumer_timeout_ms > 0

Returns a single KafkaMessage from the message iterator

Raises ConsumerTimeout after consumer_timeout_ms and no message

---

Note: This is also the method called internally during iteration

---

offsets(*group=None*)

Get internal consumer offset values

Keyword Arguments **group** – Either "fetch", "commit", "task_done", or "highwater". If no group specified, returns all groups.

Returns A copy of internal offsets struct

set_topic_partitions(**topics*)

Set the topic/partitions to consume Optionally specify offsets to start from

Accepts types:

•str (utf-8): topic name (will consume all available partitions)

•tuple: (topic, partition)

•**dict:**

- { topic: partition }
- { topic: [partition list] }
- { topic: (partition tuple,) }

Optionally, offsets can be specified directly:

•tuple: (topic, partition, offset)

•dict: { (topic, partition): offset, ... }

Example:

```
kafka = KafkaConsumer()

# Consume topic1-all; topic2-partition2; topic3-partition0
kafka.set_topic_partitions("topic1", ("topic2", 2), {"topic3": 0})

# Consume topic1-0 starting at offset 12, and topic2-1 at offset 45
# using tuples --
kafka.set_topic_partitions(("topic1", 0, 12), ("topic2", 1, 45))
```

```
# using dict --
kafka.set_topic_partitions({ ("topic1", 0): 12, ("topic2", 1): 45 })
```

task_done(*message*)
Mark a fetched message as consumed.

Offsets for messages marked as "task_done" will be stored back to the kafka cluster for this consumer group on commit()

Parameters **message** (KafkaMessage) – the message to mark as complete

Returns True, unless the topic-partition for this message has not been configured for the consumer. In normal operation, this should not happen. But see github issue 364.

kafka.partitioner package

Submodules

kafka.partitioner.base module

class kafka.partitioner.base.**Partitioner**(*partitions*)
Bases: object

Base class for a partitioner

partition(*key*, *partitions=None*)
Takes a string key and num_partitions as argument and returns a partition to be used for the message

Parameters

- **key** – the key to use for partitioning
- **partitions** – (optional) a list of partitions.

kafka.partitioner.hashed module

kafka.partitioner.hashed.**HashedPartitioner**
alias of *LegacyPartitioner*

class kafka.partitioner.hashed.**LegacyPartitioner**(*partitions*)
Bases: *kafka.partitioner.base.Partitioner*

DEPRECATED – See Issue 374

Implements a partitioner which selects the target partition based on the hash of the key

partition(*key*, *partitions=None*)

class kafka.partitioner.hashed.**Murmur2Partitioner**(*partitions*)
Bases: *kafka.partitioner.base.Partitioner*

Implements a partitioner which selects the target partition based on the hash of the key. Attempts to apply the same hashing function as mainline java client.

partition(*key*, *partitions=None*)

kafka.partitioner.hashed.**murmur2**(*key*)
Pure-python Murmur2 implementation.

Based on java client, see org.apache.kafka.common.utils.Utils.murmur2

Parameters **key** – if not a bytearray, converted via bytearray(str(key))

Returns: MurmurHash2 of key bytearray

kafka.partitioner.roundrobin module

class kafka.partitioner.roundrobin.**RoundRobinPartitioner**(*partitions*)
Bases: *kafka.partitioner.base.Partitioner*

Implements a round robin partitioner which sends data to partitions in a round robin fashion

partition(*key*, *partitions=None*)

Module contents

class kafka.partitioner.**RoundRobinPartitioner**(*partitions*)
Bases: *kafka.partitioner.base.Partitioner*

Implements a round robin partitioner which sends data to partitions in a round robin fashion

partition(*key*, *partitions=None*)

kafka.partitioner.**HashedPartitioner**
alias of *LegacyPartitioner*

class kafka.partitioner.**Murmur2Partitioner**(*partitions*)
Bases: *kafka.partitioner.base.Partitioner*

Implements a partitioner which selects the target partition based on the hash of the key. Attempts to apply the same hashing function as mainline java client.

partition(*key*, *partitions=None*)

class kafka.partitioner.**LegacyPartitioner**(*partitions*)
Bases: *kafka.partitioner.base.Partitioner*

DEPRECATED – See Issue 374

Implements a partitioner which selects the target partition based on the hash of the key

partition(*key*, *partitions=None*)

kafka.producer package

Submodules

kafka.producer.base module

class kafka.producer.base.**Producer**(*client*, *req_acks=1*, *ack_timeout=1000*, *codec=None*, *sync_fail_on_error=True*, *async=False*, *batch_send=False*, *batch_send_every_n=20*, *batch_send_every_t=20*, *async_retry_limit=None*, *async_retry_backoff_ms=100*, *async_retry_on_timeouts=True*, *async_queue_maxsize=0*, *async_queue_put_timeout=0*, *async_log_messages_on_error=True*, *async_stop_timeout=30*)

Bases: object

Base class to be used by producers

Parameters

- **client** (KafkaClient) – instance to use for broker communications. If async=True, the background thread will use client.copy(), which is expected to return a thread-safe object.
- **codec** (*kafka.protocol.ALL_CODECS*) – compression codec to use.
- **req_acks** (*int, optional*) – A value indicating the acknowledgements that the server must receive before responding to the request, defaults to 1 (local ack).
- **ack_timeout** (*int, optional*) – millisecond timeout to wait for the configured req_acks, defaults to 1000.
- **sync_fail_on_error** (*bool, optional*) – whether sync producer should raise exceptions (True), or just return errors (False), defaults to True.
- **async** (*bool, optional*) – send message using a background thread, defaults to False.
- **batch_send_every_n** (*int, optional*) – If async is True, messages are sent in batches of this size, defaults to 20.
- **batch_send_every_t** (*int or float, optional*) – If async is True, messages are sent immediately after this timeout in seconds, even if there are fewer than batch_send_every_n, defaults to 20.
- **async_retry_limit** (*int, optional*) – number of retries for failed messages or None for unlimited, defaults to None / unlimited.
- **async_retry_backoff_ms** (*int, optional*) – milliseconds to backoff on failed messages, defaults to 100.
- **async_retry_on_timeouts** (*bool, optional*) – whether to retry on RequestTimeoutError, defaults to True.
- **async_queue_maxsize** (*int, optional*) – limit to the size of the internal message queue in number of messages (not size), defaults to 0 (no limit).
- **async_queue_put_timeout** (*int or float, optional*) – timeout seconds for queue.put in send_messages for async producers – will only apply if async_queue_maxsize > 0 and the queue is Full, defaults to 0 (fail immediately on full queue).
- **async_log_messages_on_error** (*bool, optional*) – set to False and the async producer will only log hash() contents on failed produce requests, defaults to True (log full messages). Hash logging will not allow you to identify the specific message that failed, but it will allow you to match failures with retries.

- **async_stop_timeout** (*int or float, optional*) – seconds to continue attempting to send queued messages after producer.stop(), defaults to 30.

Deprecated Arguments:

batch_send (bool, optional): If True, messages are sent by a background thread in batches, defaults to False. Deprecated, use 'async'

ACK_AFTER_CLUSTER_COMMIT = -1

ACK_AFTER_LOCAL_WRITE = 1

ACK_NOT_REQUIRED = 0

DEFAULT_ACK_TIMEOUT = 1000

send_messages(*topic*, *partition*, **msg*)

Helper method to send produce requests @param: topic, name of topic for produce request – type str @param: partition, partition number for produce request – type int @param: *msg, one or more message payloads – type bytes @returns: ResponseRequest returned by server raises on error

Note that msg type *must* be encoded to bytes by user. Passing unicode message will not work, for example you should encode before calling send_messages via something like *unicode_message.encode('utf-8')*

All messages produced via this method will set the message 'key' to Null

stop(*timeout=1*)

Stop the producer. Optionally wait for the specified timeout before forcefully cleaning up.

kafka.producer.keyed module

class kafka.producer.keyed.**KeyedProducer**(**args*, ***kwargs*)

Bases: *kafka.producer.base.Producer*

A producer which distributes messages to partitions based on the key

See Producer class for Arguments

Additional Arguments:

partitioner: A partitioner class that will be used to get the partition to send the message to. Must be derived from Partitioner. Defaults to HashedPartitioner.

send(*topic*, *key*, *msg*)

send_messages(*topic*, *key*, **msg*)

kafka.producer.simple module

class kafka.producer.simple.**SimpleProducer**(**args*, ***kwargs*)

Bases: *kafka.producer.base.Producer*

A simple, round-robin producer.

See Producer class for Base Arguments

Additional Arguments:

random_start (bool, optional): randomize the initial partition which the first message block will be published to, otherwise if false, the first message block will always publish to partition 0 before cycling through each partition, defaults to True.

send_messages(*topic*, **msg*)

Module contents

class kafka.producer.**SimpleProducer**(**args*, ***kwargs*)
Bases: *kafka.producer.base.Producer*

A simple, round-robin producer.

See Producer class for Base Arguments

Additional Arguments:

random_start (bool, optional): randomize the initial partition which the first message block will be published to, otherwise if false, the first message block will always publish to partition 0 before cycling through each partition, defaults to True.

send_messages(*topic*, **msg*)

class kafka.producer.**KeyedProducer**(**args*, ***kwargs*)
Bases: *kafka.producer.base.Producer*

A producer which distributes messages to partitions based on the key

See Producer class for Arguments

Additional Arguments:

partitioner: A partitioner class that will be used to get the partition to send the message to. Must be derived from Partitioner. Defaults to HashedPartitioner.

send(*topic*, *key*, *msg*)

send_messages(*topic*, *key*, **msg*)

Submodules

kafka.client module

class kafka.client.**KafkaClient**(*hosts*, *client_id='kafka-python'*, *timeout=120*, *correlation_id=0*)
Bases: object

CLIENT_ID = 'kafka-python'

close()

copy()
Create an inactive copy of the client object, suitable for passing to a separate thread.

Note that the copied connections are not initialized, so reinit() must be called on the returned copy.

ensure_topic_exists(*topic*, *timeout=30*)

get_partition_ids_for_topic(*topic*)

has_metadata_for_topic(*topic*)

load_metadata_for_topics(**topics*)
Fetch broker and topic-partition metadata from the server, and update internal data: broker list, topic/partition list, and topic/parition -> broker map

This method should be called after receiving any error

Parameters ***topics** (*optional*) – If a list of topics is provided, the metadata refresh will be limited to the specified topics only.

If the broker is configured to not auto-create topics, expect UnknownTopicOrPartitionError for topics that don't exist

If the broker is configured to auto-create topics, expect LeaderNotAvailableError for new topics until partitions have been initialized.

Exceptions *will not* be raised in a full refresh (i.e. no topic list) In this case, error codes will be logged as errors

Partition-level errors will also not be raised here (a single partition w/o a leader, for example)

reinit()

reset_all_metadata()

reset_topic_metadata(**topics*)

send_fetch_request(*payloads=[]*, *fail_on_error=True*, *callback=None*, *max_wait_time=100*, *min_bytes=4096*)

Encode and send a FetchRequest

Payloads are grouped by topic and partition so they can be pipelined to the same brokers.

send_metadata_request(*payloads=[]*, *fail_on_error=True*, *callback=None*)

send_offset_commit_request(*group*, *payloads=[]*, *fail_on_error=True*, *callback=None*)

send_offset_fetch_request(*group*, *payloads=[]*, *fail_on_error=True*, *callback=None*)

send_offset_request(*payloads=[]*, *fail_on_error=True*, *callback=None*)

send_produce_request(*payloads=[]*, *acks=1*, *timeout=1000*, *fail_on_error=True*, *callback=None*)

Encode and send some ProduceRequests

ProduceRequests will be grouped by (topic, partition) and then sent to a specific broker. Output is a list of responses in the same order as the list of payloads specified

Parameters

- **payloads** (*list of ProduceRequest*) – produce requests to send to kafka ProduceRequest payloads must not contain duplicates for any topic-partition.
- **acks** (*int, optional*) – how many acks the servers should receive from replica brokers before responding to the request. If it is 0, the server will not send any response. If it is 1, the server will wait until the data is written to the local log before sending a response. If it is -1, the server will wait until the message is committed by all in-sync replicas before sending a response. For any value > 1, the server will wait for this number of acks to occur (but the server will never wait for more acknowledgements than there are in-sync replicas). defaults to 1.
- **timeout** (*int, optional*) – maximum time in milliseconds the server can await the receipt of the number of acks, defaults to 1000.
- **fail_on_error** (*bool, optional*) – raise exceptions on connection and server response errors, defaults to True.
- **callback** (*function, optional*) – instead of returning the ProduceResponse, first pass it through this function, defaults to None.

Returns list of ProduceResponses, or callback results if supplied, in the order of input payloads

topics

kafka.codec module

kafka.codec.**gzip_decode**(*payload*)

kafka.codec.**gzip_encode**(*payload*)

kafka.codec.**has_gzip**()

kafka.codec.**has_snappy**()

kafka.codec.**snappy_decode**(*payload*)

kafka.codec.**snappy_encode**(*payload*, *xerial_compatible=False*, *xerial_blocksize=32768*)

Encodes the given data with snappy if xerial_compatible is set then the stream is encoded in a fashion compatible with the xerial snappy library

The block size (xerial_blocksize) controls how frequent the blocking occurs 32k is the default in the xerial library.

The format winds up being

Header	Block1 len	Block1 data	Blockn len	Blockn d
	BE int32	snappy bytes	BE int32	snappy b

It is important to not that the blocksize is the amount of uncompressed data presented to snappy at each block, whereas the blocklen is the number of bytes that will be present in the stream, that is the length will always be <= blocksize.

kafka.common module

exception kafka.common.**AsyncProducerQueueFull**(*failed_msgs*, **args*)

Bases: *kafka.common.KafkaError*

class kafka.common.**BrokerMetadata**(*nodeId*, *host*, *port*)

Bases: tuple

host

Alias for field number 1

nodeId

Alias for field number 0

port

Alias for field number 2

exception kafka.common.**BrokerNotAvailableError**

Bases: *kafka.common.BrokerResponseError*

errno = 8

message = 'BROKER_NOT_AVAILABLE'

exception kafka.common.**BrokerResponseError**

Bases: *kafka.common.KafkaError*

exception kafka.common.**BufferUnderflowError**

Bases: *kafka.common.KafkaError*

exception kafka.common.**ChecksumError**

Bases: *kafka.common.KafkaError*

exception kafka.common.**ConnectionError**

Bases: *kafka.common.KafkaError*

exception kafka.common.**ConsumerFetchSizeTooSmall**
Bases: *kafka.common.KafkaError*

exception kafka.common.**ConsumerNoMoreData**
Bases: *kafka.common.KafkaError*

exception kafka.common.**ConsumerTimeout**
Bases: *kafka.common.KafkaError*

exception kafka.common.**FailedPayloadsError**(*payload*, **args*)
Bases: *kafka.common.KafkaError*

class kafka.common.**FetchRequest**(*topic*, *partition*, *offset*, *max_bytes*)
Bases: tuple

max_bytes
Alias for field number 3

offset
Alias for field number 2

partition
Alias for field number 1

topic
Alias for field number 0

class kafka.common.**FetchResponse**(*topic*, *partition*, *error*, *highwaterMark*, *messages*)
Bases: tuple

error
Alias for field number 2

highwaterMark
Alias for field number 3

messages
Alias for field number 4

partition
Alias for field number 1

topic
Alias for field number 0

exception kafka.common.**InvalidFetchRequestError**
Bases: *kafka.common.BrokerResponseError*

errno = 4

message = 'INVALID_FETCH_SIZE'

exception kafka.common.**InvalidMessageError**
Bases: *kafka.common.BrokerResponseError*

errno = 2

message = 'INVALID_MESSAGE'

exception kafka.common.**KafkaConfigurationError**
Bases: *kafka.common.KafkaError*

exception kafka.common.**KafkaError**
Bases: exceptions.RuntimeError

class kafka.common.**KafkaMessage**(*topic*, *partition*, *offset*, *key*, *value*)
Bases: tuple

key
Alias for field number 3

offset
Alias for field number 2

partition
Alias for field number 1

topic
Alias for field number 0

value
Alias for field number 4

exception kafka.common.**KafkaTimeoutError**
Bases: *kafka.common.KafkaError*

exception kafka.common.**KafkaUnavailableError**
Bases: *kafka.common.KafkaError*

exception kafka.common.**LeaderNotAvailableError**
Bases: *kafka.common.BrokerResponseError*

errno = 5

message = 'LEADER_NOT_AVAILABLE'

class kafka.common.**Message**(*magic*, *attributes*, *key*, *value*)
Bases: tuple

attributes
Alias for field number 1

key
Alias for field number 2

magic
Alias for field number 0

value
Alias for field number 3

exception kafka.common.**MessageSizeTooLargeError**
Bases: *kafka.common.BrokerResponseError*

errno = 10

message = 'MESSAGE_SIZE_TOO_LARGE'

class kafka.common.**MetadataRequest**(*topics*)
Bases: tuple

topics
Alias for field number 0

class kafka.common.**MetadataResponse**(*brokers*, *topics*)
Bases: tuple

brokers
Alias for field number 0

topics
Alias for field number 1

exception kafka.common.**NotLeaderForPartitionError**
Bases: *kafka.common.BrokerResponseError*

errno = 6

message = 'NOT_LEADER_FOR_PARTITION'

class kafka.common.**OffsetAndMessage**(*offset*, *message*)
Bases: tuple

message
Alias for field number 1

offset
Alias for field number 0

class kafka.common.**OffsetCommitRequest**(*topic*, *partition*, *offset*, *metadata*)
Bases: tuple

metadata
Alias for field number 3

offset
Alias for field number 2

partition
Alias for field number 1

topic
Alias for field number 0

class kafka.common.**OffsetCommitResponse**(*topic*, *partition*, *error*)
Bases: tuple

error
Alias for field number 2

partition
Alias for field number 1

topic
Alias for field number 0

class kafka.common.**OffsetFetchRequest**(*topic*, *partition*)
Bases: tuple

partition
Alias for field number 1

topic
Alias for field number 0

class kafka.common.**OffsetFetchResponse**(*topic*, *partition*, *offset*, *metadata*, *error*)
Bases: tuple

error
Alias for field number 4

metadata
Alias for field number 3

offset
Alias for field number 2

partition
Alias for field number 1

topic
Alias for field number 0

exception kafka.common.**OffsetMetadataTooLargeError**
Bases: *kafka.common.BrokerResponseError*

errno = 12

message = 'OFFSET_METADATA_TOO_LARGE'

exception kafka.common.**OffsetOutOfRangeError**
Bases: *kafka.common.BrokerResponseError*

errno = 1

message = 'OFFSET_OUT_OF_RANGE'

class kafka.common.**OffsetRequest**(*topic*, *partition*, *time*, *max_offsets*)
Bases: tuple

max_offsets
Alias for field number 3

partition
Alias for field number 1

time
Alias for field number 2

topic
Alias for field number 0

class kafka.common.**OffsetResponse**(*topic*, *partition*, *error*, *offsets*)
Bases: tuple

error
Alias for field number 2

offsets
Alias for field number 3

partition
Alias for field number 1

topic
Alias for field number 0

class kafka.common.**PartitionMetadata**(*topic*, *partition*, *leader*, *replicas*, *isr*, *error*)
Bases: tuple

error
Alias for field number 5

isr
Alias for field number 4

leader
Alias for field number 2

partition
Alias for field number 1

replicas
Alias for field number 3

topic
Alias for field number 0

class kafka.common.**ProduceRequest**(*topic*, *partition*, *messages*)
Bases: tuple

messages
Alias for field number 2

partition
Alias for field number 1

topic
Alias for field number 0

class kafka.common.**ProduceResponse**(*topic*, *partition*, *error*, *offset*)
Bases: tuple

error
Alias for field number 2

offset
Alias for field number 3

partition
Alias for field number 1

topic
Alias for field number 0

exception kafka.common.**ProtocolError**
Bases: *kafka.common.KafkaError*

exception kafka.common.**ReplicaNotAvailableError**
Bases: *kafka.common.BrokerResponseError*

errno = 9

message = 'REPLICA_NOT_AVAILABLE'

exception kafka.common.**RequestTimedOutError**
Bases: *kafka.common.BrokerResponseError*

errno = 7

message = 'REQUEST_TIMED_OUT'

class kafka.common.**RetryOptions**(*limit*, *backoff_ms*, *retry_on_timeouts*)
Bases: tuple

backoff_ms
Alias for field number 1

limit
Alias for field number 0

retry_on_timeouts
Alias for field number 2

exception kafka.common.**StaleControllerEpochError**
Bases: *kafka.common.BrokerResponseError*

errno = 11

message = 'STALE_CONTROLLER_EPOCH'

exception kafka.common.**StaleLeaderEpochCodeError**
Bases: *kafka.common.BrokerResponseError*

errno = 13

message = 'STALE_LEADER_EPOCH_CODE'

class kafka.common.**TopicAndPartition**(*topic*, *partition*)
Bases: tuple

partition
Alias for field number 1

topic
Alias for field number 0

class kafka.common.**TopicMetadata**(*topic*, *error*, *partitions*)
Bases: tuple

error
Alias for field number 1

partitions
Alias for field number 2

topic
Alias for field number 0

exception kafka.common.**UnknownError**
Bases: *kafka.common.BrokerResponseError*

errno = -1

message = 'UNKNOWN'

exception kafka.common.**UnknownTopicOrPartitionError**
Bases: *kafka.common.BrokerResponseError*

errno = 3

message = 'UNKNOWN_TOPIC_OR_PARTITON'

exception kafka.common.**UnsupportedCodecError**
Bases: *kafka.common.KafkaError*

kafka.common.**check_error**(*response*)

kafka.common.**x**
alias of *UnknownTopicOrPartitionError*

kafka.conn module

class kafka.conn.**KafkaConnection**(*host*, *port*, *timeout=120*)
Bases: thread._local

A socket connection to a single Kafka broker

This class is _not_ thread safe. Each call to *send* must be followed by a call to *recv* in order to get the correct response. Eventually, we can do something in here to facilitate multiplexed requests/responses since the Kafka API includes a correlation id.

Parameters

- **host** – the host name or IP address of a kafka broker
- **port** – the port number the kafka broker is listening on
- **timeout** – default 120. The socket timeout for sending and receiving data in seconds. None means no timeout, so a request can block forever.

close()
Shutdown and close the connection socket

copy()
Create an inactive copy of the connection object, suitable for passing to a background thread.

The returned copy is not connected; you must call reinit() before using.

recv(*request_id*)
Get a response packet from Kafka

Parameters **request_id** – can be any int (only used for debug logging...)

Returns Encoded kafka packet response from server

Return type str

reinit()
Re-initialize the socket connection close current socket (if open) and start a fresh connection raise ConnectionError on error

send(*request_id*, *payload*)
Send a request to Kafka

Arguments:: request_id (int): can be any int (used only for debug logging...) payload: an encoded kafka packet (see KafkaProtocol)

kafka.conn.**collect_hosts**(*hosts*, *randomize=True*)
Collects a comma-separated set of hosts (host:port) and optionally randomize the returned list.

kafka.context module

Context manager to commit/rollback consumer offsets.

class kafka.context.**OffsetCommitContext**(*consumer*)
Bases: object

Provides commit/rollback semantics around a *SimpleConsumer*.

Usage assumes that *auto_commit* is disabled, that messages are consumed in batches, and that the consuming process will record its own successful processing of each message. Both the commit and rollback operations respect a "high-water mark" to ensure that last unsuccessfully processed message will be retried.

Example:

```
consumer = SimpleConsumer(client, group, topic, auto_commit=False)
consumer.provide_partition_info()
consumer.fetch_last_known_offsets()

while some_condition:
```

```
    with OffsetCommitContext(consumer) as context:
        messages = consumer.get_messages(count, block=False)

        for partition, message in messages:
            if can_process(message):
                context.mark(partition, message.offset)
            else:
                break

        if not context:
            sleep(delay)
```

These semantics allow for deferred message processing (e.g. if *can_process* compares message time to clock time) and for repeated processing of the last unsuccessful message (until some external error is resolved).

commit()
Commit this context's offsets:

- If the high-water mark has moved, commit up to and position the consumer at the high-water mark.
- Otherwise, reset to the consumer to the initial offsets.

commit_partition_offsets(*partition_offsets*)
Commit explicit partition/offset pairs.

handle_out_of_range()
Handle out of range condition by seeking to the beginning of valid ranges.

This assumes that an out of range doesn't happen by seeking past the end of valid ranges – which is far less likely.

mark(*partition*, *offset*)
Set the high-water mark in the current context.

In order to know the current partition, it is helpful to initialize the consumer to provide partition info via:

```
consumer.provide_partition_info()
```

rollback()
Rollback this context:

- Position the consumer at the initial offsets.

update_consumer_offsets(*partition_offsets*)
Update consumer offsets to explicit positions.

kafka.protocol module

class kafka.protocol.**KafkaProtocol**
Bases: object

Class to encapsulate all of the protocol encoding/decoding. This class does not have any state associated with it, it is purely for organization.

FETCH_KEY = 1

METADATA_KEY = 3

OFFSET_COMMIT_KEY = 8

OFFSET_FETCH_KEY = 9

OFFSET_KEY = 2

PRODUCE_KEY = 0

classmethod decode_fetch_response(*data*)
Decode bytes to a FetchResponse

Parameters **data** – bytes to decode

classmethod decode_metadata_response(*data*)
Decode bytes to a MetadataResponse

Parameters **data** – bytes to decode

classmethod decode_offset_commit_response(*data*)
Decode bytes to an OffsetCommitResponse

Parameters **data** – bytes to decode

classmethod decode_offset_fetch_response(*data*)
Decode bytes to an OffsetFetchResponse

Parameters **data** – bytes to decode

classmethod decode_offset_response(*data*)
Decode bytes to an OffsetResponse

Parameters **data** – bytes to decode

classmethod decode_produce_response(*data*)
Decode bytes to a ProduceResponse

Parameters **data** – bytes to decode

classmethod encode_fetch_request(*client_id*, *correlation_id*, *payloads=None*, *max_wait_time=100*, *min_bytes=4096*)
Encodes some FetchRequest structs

Parameters

- **client_id** – string
- **correlation_id** – int
- **payloads** – list of FetchRequest
- **max_wait_time** – int, how long to block waiting on min_bytes of data
- **min_bytes** – int, the minimum number of bytes to accumulate before returning the response

classmethod encode_metadata_request(*client_id*, *correlation_id*, *topics=None*, *payloads=None*)
Encode a MetadataRequest

Parameters

- **client_id** – string
- **correlation_id** – int
- **topics** – list of strings

classmethod encode_offset_commit_request(*client_id*, *correlation_id*, *group*, *payloads*)
Encode some OffsetCommitRequest structs

Parameters

- **client_id** – string
- **correlation_id** – int
- **group** – string, the consumer group you are committing offsets for
- **payloads** – list of OffsetCommitRequest

classmethod encode_offset_fetch_request(*client_id*, *correlation_id*, *group*, *payloads*)

Encode some OffsetFetchRequest structs

Parameters

- **client_id** – string
- **correlation_id** – int
- **group** – string, the consumer group you are fetching offsets for
- **payloads** – list of OffsetFetchRequest

classmethod encode_offset_request(*client_id*, *correlation_id*, *payloads=None*)

classmethod encode_produce_request(*client_id*, *correlation_id*, *payloads=None*, *acks=1*, *timeout=1000*)

Encode some ProduceRequest structs

Parameters

- **client_id** – string
- **correlation_id** – int
- **payloads** – list of ProduceRequest
- **acks** – How "acky" you want the request to be 0: immediate response 1: written to disk by the leader 2+: waits for this many number of replicas to sync -1: waits for all replicas to be in sync
- **timeout** – Maximum time the server will wait for acks from replicas. This is _not_ a socket timeout

kafka.protocol.**create_gzip_message**(*payloads*, *key=None*)

Construct a Gzipped Message containing multiple Messages

The given payloads will be encoded, compressed, and sent as a single atomic message to Kafka.

Parameters

- **payloads** – list(bytes), a list of payload to send be sent to Kafka
- **key** – bytes, a key used for partition routing (optional)

kafka.protocol.**create_message**(*payload*, *key=None*)

Construct a Message

Parameters

- **payload** – bytes, the payload to send to Kafka
- **key** – bytes, a key used for partition routing (optional)

kafka.protocol.**create_message_set**(*messages*, *codec=0*, *key=None*)

Create a message set using the given codec.

If codec is CODEC_NONE, return a list of raw Kafka messages. Otherwise, return a list containing a single codec-encoded message.

kafka.protocol.**create_snappy_message**(*payloads*, *key=None*)
Construct a Snappy Message containing multiple Messages

The given payloads will be encoded, compressed, and sent as a single atomic message to Kafka.

Parameters

- **payloads** – list(bytes), a list of payload to send be sent to Kafka
- **key** – bytes, a key used for partition routing (optional)

kafka.util module

class kafka.util.**ReentrantTimer**(*t*, *fn*, **args*, ***kwargs*)
Bases: object

A timer that can be restarted, unlike threading.Timer (although this uses threading.Timer)

Parameters

- **t** – timer interval in milliseconds
- **fn** – a callable to invoke
- **args** – tuple of args to be passed to function
- **kwargs** – keyword arguments to be passed to function

start()

stop()

kafka.util.**crc32**(*data*)

kafka.util.**group_by_topic_and_partition**(*tuples*)

kafka.util.**kafka_bytestring**(*s*)
Takes a string or bytes instance Returns bytes, encoding strings in utf-8 as necessary

kafka.util.**read_int_string**(*data*, *cur*)

kafka.util.**read_short_string**(*data*, *cur*)

kafka.util.**relative_unpack**(*fmt*, *data*, *cur*)

kafka.util.**write_int_string**(*s*)

kafka.util.**write_short_string**(*s*)

Module contents

class kafka.**KafkaClient**(*hosts*, *client_id='kafka-python'*, *timeout=120*, *correlation_id=0*)
Bases: object

CLIENT_ID = 'kafka-python'

close()

copy()
Create an inactive copy of the client object, suitable for passing to a separate thread.

Note that the copied connections are not initialized, so reinit() must be called on the returned copy.

ensure_topic_exists(*topic*, *timeout=30*)

get_partition_ids_for_topic(*topic*)

has_metadata_for_topic(*topic*)

load_metadata_for_topics(**topics*)
Fetch broker and topic-partition metadata from the server, and update internal data: broker list, topic/partition list, and topic/parition -> broker map

This method should be called after receiving any error

Parameters ***topics** (*optional*) – If a list of topics is provided, the metadata refresh will be limited to the specified topics only.

If the broker is configured to not auto-create topics, expect UnknownTopicOrPartitionError for topics that don't exist

If the broker is configured to auto-create topics, expect LeaderNotAvailableError for new topics until partitions have been initialized.

Exceptions *will not* be raised in a full refresh (i.e. no topic list) In this case, error codes will be logged as errors

Partition-level errors will also not be raised here (a single partition w/o a leader, for example)

reinit()

reset_all_metadata()

reset_topic_metadata(**topics*)

send_fetch_request(*payloads=[]*, *fail_on_error=True*, *callback=None*, *max_wait_time=100*, *min_bytes=4096*)
Encode and send a FetchRequest

Payloads are grouped by topic and partition so they can be pipelined to the same brokers.

send_metadata_request(*payloads=[]*, *fail_on_error=True*, *callback=None*)

send_offset_commit_request(*group*, *payloads=[]*, *fail_on_error=True*, *callback=None*)

send_offset_fetch_request(*group*, *payloads=[]*, *fail_on_error=True*, *callback=None*)

send_offset_request(*payloads=[]*, *fail_on_error=True*, *callback=None*)

send_produce_request(*payloads=[]*, *acks=1*, *timeout=1000*, *fail_on_error=True*, *callback=None*)
Encode and send some ProduceRequests

ProduceRequests will be grouped by (topic, partition) and then sent to a specific broker. Output is a list of responses in the same order as the list of payloads specified

Parameters

- **payloads** (*list of ProduceRequest*) – produce requests to send to kafka ProduceRequest payloads must not contain duplicates for any topic-partition.
- **acks** (*int, optional*) – how many acks the servers should receive from replica brokers before responding to the request. If it is 0, the server will not send any response. If it is 1, the server will wait until the data is written to the local log before sending a response. If it is -1, the server will wait until the message is committed by all in-sync replicas before sending a response. For any value > 1, the server will wait for this number of acks to occur (but the server will never wait for more acknowledgements than there are in-sync replicas). defaults to 1.
- **timeout** (*int, optional*) – maximum time in milliseconds the server can await the receipt of the number of acks, defaults to 1000.

- **fail_on_error** (*bool, optional*) – raise exceptions on connection and server response errors, defaults to True.
- **callback** (*function, optional*) – instead of returning the ProduceResponse, first pass it through this function, defaults to None.

Returns list of ProduceResponses, or callback results if supplied, in the order of input payloads

topics

class kafka.**KafkaConnection**(*host*, *port*, *timeout=120*)
Bases: thread._local

A socket connection to a single Kafka broker

This class is _not_ thread safe. Each call to *send* must be followed by a call to *recv* in order to get the correct response. Eventually, we can do something in here to facilitate multiplexed requests/responses since the Kafka API includes a correlation id.

Parameters

- **host** – the host name or IP address of a kafka broker
- **port** – the port number the kafka broker is listening on
- **timeout** – default 120. The socket timeout for sending and receiving data in seconds. None means no timeout, so a request can block forever.

close()
Shutdown and close the connection socket

copy()
Create an inactive copy of the connection object, suitable for passing to a background thread.

The returned copy is not connected; you must call reinit() before using.

recv(*request_id*)
Get a response packet from Kafka

Parameters **request_id** – can be any int (only used for debug logging...)

Returns Encoded kafka packet response from server

Return type str

reinit()
Re-initialize the socket connection close current socket (if open) and start a fresh connection raise ConnectionError on error

send(*request_id*, *payload*)
Send a request to Kafka

Arguments:: request_id (int): can be any int (used only for debug logging...) payload: an encoded kafka packet (see KafkaProtocol)

class kafka.**SimpleProducer**(**args*, ***kwargs*)
Bases: *kafka.producer.base.Producer*

A simple, round-robin producer.

See Producer class for Base Arguments

Additional Arguments:

random_start (bool, optional): randomize the initial partition which the first message block will be published to, otherwise if false, the first message block will always publish to partition 0 before cycling through each partition, defaults to True.

send_messages(*topic*, **msg*)

class kafka.**KeyedProducer**(**args*, ***kwargs*)
Bases: *kafka.producer.base.Producer*

A producer which distributes messages to partitions based on the key

See Producer class for Arguments

Additional Arguments:

partitioner: A partitioner class that will be used to get the partition to send the message to. Must be derived from Partitioner. Defaults to HashedPartitioner.

send(*topic*, *key*, *msg*)

send_messages(*topic*, *key*, **msg*)

class kafka.**RoundRobinPartitioner**(*partitions*)
Bases: *kafka.partitioner.base.Partitioner*

Implements a round robin partitioner which sends data to partitions in a round robin fashion

partition(*key*, *partitions=None*)

kafka.**HashedPartitioner**
alias of LegacyPartitioner

class kafka.**SimpleConsumer**(*client*, *group*, *topic*, *auto_commit=True*, *partitions=None*, *auto_commit_every_n=100*, *auto_commit_every_t=5000*, *fetch_size_bytes=4096*, *buffer_size=4096*, *max_buffer_size=32768*, *iter_timeout=None*, *auto_offset_reset='largest'*)
Bases: *kafka.consumer.base.Consumer*

A simple consumer implementation that consumes all/specified partitions for a topic

Parameters

- **client** – a connected KafkaClient
- **group** – a name for this consumer, used for offset storage and must be unique If you are connecting to a server that does not support offset commit/fetch (any prior to 0.8.1.1), then you *must* set this to None
- **topic** – the topic to consume

Keyword Arguments

- **partitions** – An optional list of partitions to consume the data from
- **auto_commit** – default True. Whether or not to auto commit the offsets
- **auto_commit_every_n** – default 100. How many messages to consume before a commit
- **auto_commit_every_t** – default 5000. How much time (in milliseconds) to wait before commit
- **fetch_size_bytes** – number of bytes to request in a FetchRequest
- **buffer_size** – default 4K. Initial number of bytes to tell kafka we have available. This will double as needed.

- **max_buffer_size** – default 16K. Max number of bytes to tell kafka we have available. None means no limit.
- **iter_timeout** – default None. How much time (in seconds) to wait for a message in the iterator before exiting. None means no timeout, so it will wait forever.
- **auto_offset_reset** – default largest. Reset partition offsets upon OffsetOutOfRangeError. Valid values are largest and smallest. Otherwise, do not reset the offsets and raise OffsetOutOfRangeError.

Auto commit details: If both auto_commit_every_n and auto_commit_every_t are set, they will reset one another when one is triggered. These triggers simply call the commit method on this class. A manual call to commit will also reset these triggers

get_message(*block=True*, *timeout=0.1*, *get_partition_info=None*)

get_messages(*count=1*, *block=True*, *timeout=0.1*)

Fetch the specified number of messages

Keyword Arguments

- **count** – Indicates the maximum number of messages to be fetched
- **block** – If True, the API will block till some messages are fetched.
- **timeout** – If block is True, the function will block for the specified time (in seconds) until count messages is fetched. If None, it will block forever.

provide_partition_info()

Indicates that partition info must be returned by the consumer

reset_partition_offset(*partition*)

Update offsets using auto_offset_reset policy (smallest|largest)

Parameters **partition** (*int*) – the partition for which offsets should be updated

Returns: Updated offset on success, None on failure

seek(*offset*, *whence*)

Alter the current offset in the consumer, similar to fseek

Parameters

- **offset** – how much to modify the offset
- **whence** – where to modify it from
  - 0 is relative to the earliest available offset (head)
  - 1 is relative to the current offset
  - 2 is relative to the latest known offset (tail)

class kafka.**MultiProcessConsumer**(*client*, *group*, *topic*, *partitions=None*, *auto_commit=True*, *auto_commit_every_n=100*, *auto_commit_every_t=5000*, *num_procs=1*, *partitions_per_proc=0*, ***simple_consumer_options*)

Bases: *kafka.consumer.base.Consumer*

A consumer implementation that consumes partitions for a topic in parallel using multiple processes

Parameters

- **client** – a connected KafkaClient

- **group** – a name for this consumer, used for offset storage and must be unique If you are connecting to a server that does not support offset commit/fetch (any prior to 0.8.1.1), then you *must* set this to None
- **topic** – the topic to consume

Keyword Arguments

- **partitions** – An optional list of partitions to consume the data from
- **auto_commit** – default True. Whether or not to auto commit the offsets
- **auto_commit_every_n** – default 100. How many messages to consume before a commit
- **auto_commit_every_t** – default 5000. How much time (in milliseconds) to wait before commit
- **num_procs** – Number of processes to start for consuming messages. The available partitions will be divided among these processes
- **partitions_per_proc** – Number of partitions to be allocated per process (overrides num_procs)

Auto commit details: If both auto_commit_every_n and auto_commit_every_t are set, they will reset one another when one is triggered. These triggers simply call the commit method on this class. A manual call to commit will also reset these triggers

get_messages(*count=1*, *block=True*, *timeout=10*)
Fetch the specified number of messages

Keyword Arguments

- **count** – Indicates the maximum number of messages to be fetched
- **block** – If True, the API will block till some messages are fetched.
- **timeout** – If block is True, the function will block for the specified time (in seconds) until count messages is fetched. If None, it will block forever.

stop()

kafka.**create_message**(*payload*, *key=None*)
Construct a Message

Parameters

- **payload** – bytes, the payload to send to Kafka
- **key** – bytes, a key used for partition routing (optional)

kafka.**create_gzip_message**(*payloads*, *key=None*)
Construct a Gzipped Message containing multiple Messages

The given payloads will be encoded, compressed, and sent as a single atomic message to Kafka.

Parameters

- **payloads** – list(bytes), a list of payload to send be sent to Kafka
- **key** – bytes, a key used for partition routing (optional)

kafka.**create_snappy_message**(*payloads*, *key=None*)
Construct a Snappy Message containing multiple Messages

The given payloads will be encoded, compressed, and sent as a single atomic message to Kafka.

Parameters

- **payloads** – list(bytes), a list of payload to send be sent to Kafka
- **key** – bytes, a key used for partition routing (optional)

class kafka.**KafkaConsumer**(**topics*, ***configs*)
Bases: object

A simpler kafka consumer

commit()
Store consumed message offsets (marked via task_done()) to kafka cluster for this consumer_group.

Returns True on success, or False if no offsets were found for commit

---

Note: this functionality requires server version >=0.8.1.1 https://cwiki.apache.org/confluence/display/KAFKA/A+Guide+To+The+Kafka+Protocol#AGuideToTheKafkaProtocol-OffsetCommit/FetchAPI

---

configure(***configs*)
Configure the consumer instance

Configuration settings can be passed to constructor, otherwise defaults will be used:

Keyword Arguments

- **bootstrap_servers** (*list*) – List of initial broker nodes the consumer should contact to bootstrap initial cluster metadata. This does not have to be the full node list. It just needs to have at least one broker that will respond to a Metadata API Request.
- **client_id** (*str*) – a unique name for this client. Defaults to 'kafka.consumer.kafka'.
- **group_id** (*str*) – the name of the consumer group to join, Offsets are fetched / committed to this group name.
- **fetch_message_max_bytes** (*int, optional*) – Maximum bytes for each topic/partition fetch request. Defaults to 1024*1024.
- **fetch_min_bytes** (*int, optional*) – Minimum amount of data the server should return for a fetch request, otherwise wait up to fetch_wait_max_ms for more data to accumulate. Defaults to 1.
- **fetch_wait_max_ms** (*int, optional*) – Maximum time for the server to block waiting for fetch_min_bytes messages to accumulate. Defaults to 100.
- **refresh_leader_backoff_ms** (*int, optional*) – Milliseconds to backoff when refreshing metadata on errors (subject to random jitter). Defaults to 200.
- **socket_timeout_ms** (*int, optional*) – TCP socket timeout in milliseconds. Defaults to 30*1000.
- **auto_offset_reset** (*str, optional*) – A policy for resetting offsets on OffsetOutOfRange errors. 'smallest' will move to the oldest available message, 'largest' will move to the most recent. Any ofther value will raise the exception. Defaults to 'largest'.
- **deserializer_class** (*callable, optional*) – Any callable that takes a raw message value and returns a deserialized value. Defaults to

  lambda msg: msg.
- **auto_commit_enable** (*bool, optional*) – Enabling auto-commit will cause the KafkaConsumer to periodically commit offsets without an explicit call to commit(). Defaults to False.

- **auto_commit_interval_ms** (*int, optional*) – If auto_commit_enabled, the milliseconds between automatic offset commits. Defaults to 60 * 1000.
- **auto_commit_interval_messages** (*int, optional*) – If auto_commit_enabled, a number of messages consumed between automatic offset commits. Defaults to None (disabled).
- **consumer_timeout_ms** (*int, optional*) – number of millisecond to throw a timeout exception to the consumer if no message is available for consumption. Defaults to -1 (dont throw exception).

Configuration parameters are described in more detail at http://kafka.apache.org/documentation.html#highlevelconsumerapi

fetch_messages()

Sends FetchRequests for all topic/partitions set for consumption

Returns Generator that yields KafkaMessage structs after deserializing with the configured *deserializer_class*

Note: Refreshes metadata on errors, and resets fetch offset on OffsetOutOfRange, per the configured *auto_offset_reset* policy

---

See also:

Key KafkaConsumer configuration parameters: * *fetch_message_max_bytes* * *fetch_max_wait_ms* * *fetch_min_bytes* * *deserializer_class* * *auto_offset_reset*

get_partition_offsets(*topic*, *partition*, *request_time_ms*, *max_num_offsets*)

Request available fetch offsets for a single topic/partition

Keyword Arguments

- **topic** (*str*) – topic for offset request
- **partition** (*int*) – partition for offset request
- **request_time_ms** (*int*) – Used to ask for all messages before a certain time (ms). There are two special values. Specify -1 to receive the latest offset (i.e. the offset of the next coming message) and -2 to receive the earliest available offset. Note that because offsets are pulled in descending order, asking for the earliest offset will always return you a single element.
- **max_num_offsets** (*int*) – Maximum offsets to include in the OffsetResponse

Returns a list of offsets in the OffsetResponse submitted for the provided topic / partition. See: https://cwiki.apache.org/confluence/display/KAFKA/A+Guide+To+The+Kafka+Protocol#AGuideToTheKafkaProtocol-OffsetAPI

next()

Return the next available message

Blocks indefinitely unless consumer_timeout_ms > 0

Returns a single KafkaMessage from the message iterator

Raises ConsumerTimeout after consumer_timeout_ms and no message

---

Note: This is also the method called internally during iteration

---

offsets(*group=None*)
Get internal consumer offset values

Keyword Arguments **group** – Either "fetch", "commit", "task_done", or "highwater". If no group specified, returns all groups.

Returns A copy of internal offsets struct

set_topic_partitions(**topics*)
Set the topic/partitions to consume Optionally specify offsets to start from

Accepts types:

•str (utf-8): topic name (will consume all available partitions)

•tuple: (topic, partition)

•**dict:**

- { topic: partition }
- { topic: [partition list] }
- { topic: (partition tuple,) }

Optionally, offsets can be specified directly:

•tuple: (topic, partition, offset)

•dict: { (topic, partition): offset, ... }

Example:

```
kafka = KafkaConsumer()

# Consume topic1-all; topic2-partition2; topic3-partition0
kafka.set_topic_partitions("topic1", ("topic2", 2), {"topic3": 0})

# Consume topic1-0 starting at offset 12, and topic2-1 at offset 45
# using tuples --
kafka.set_topic_partitions(("topic1", 0, 12), ("topic2", 1, 45))

# using dict --
kafka.set_topic_partitions({ ("topic1", 0): 12, ("topic2", 1): 45 })
```

task_done(*message*)
Mark a fetched message as consumed.

Offsets for messages marked as "task_done" will be stored back to the kafka cluster for this consumer group on commit()

Parameters **message** (KafkaMessage) – the message to mark as complete

Returns True, unless the topic-partition for this message has not been configured for the consumer. In normal operation, this should not happen. But see github issue 364.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

k

Index

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X