pykafka

Release 2.8.0-dev.3

Aug 15, 2018

Contents

1 Getting Started 3

2 Using the librdkafka extension 5

3 Operational Tools 7

4 PyKafka or kafka-python? 9

5 Contributing 11

6 Support 13

Python Module Index 75

PyKafka is a programmer-friendly Kafka client for Python. It includes Python implementations of Kafka producers and consumers, which are optionally backed by a C extension built on librdkafka. It runs under Python 2.7+, Python 3.4+, and PyPy, and supports versions of Kafka 0.8.2 and newer.

PyKafka's primary goal is to provide a similar level of abstraction to the JVM Kafka client using idioms familiar to Python programmers and exposing the most Pythonic API possible.

You can install PyKafka from PyPI with

```
$ pip install pykafka
```

or from conda-forge with

```
$ conda install -c conda-forge pykafka
```

Full documentation and usage examples for PyKafka can be found on readthedocs.

You can install PyKafka for local development and testing by cloning this repository and running

```
$ python setup.py develop
```

CHAPTER 1

Getting Started

Assuming you have at least one Kafka instance running on localhost, you can use PyKafka to connect to it.

```
>>> from pykafka import KafkaClient
>>> client = KafkaClient(hosts="127.0.0.1:9092,127.0.0.1:9093,...")
```

Or, for a TLS connection, you might write (and also see `SslConfig` docs for further details):

```
>>> from pykafka import KafkaClient, SslConfig
>>> config = SslConfig(cafile='/your/ca.cert',
...                    certfile='/your/client.cert',  # optional
...                    keyfile='/your/client.key',  # optional
...                    password='unlock my client key please')  # optional
>>> client = KafkaClient(hosts="127.0.0.1:<ssl-port>,...",
...                      ssl_config=config)
```

If the cluster you've connected to has any topics defined on it, you can list them with:

```
>>> client.topics
>>> topic = client.topics['my.test']
```

Once you've got a *Topic*, you can create a *Producer* for it and start producing messages.

```
>>> with topic.get_sync_producer() as producer:
...     for i in range(4):
...         producer.produce('test message ' + str(i ** 2))
```

The example above would produce to kafka synchronously - the call only returns after we have confirmation that the message made it to the cluster.

To achieve higher throughput, we recommend using the `Producer` in asynchronous mode, so that `produce()` calls will return immediately and the producer may opt to send messages in larger batches. The `Producer` collects produced messages in an internal queue for `linger_ms` before sending each batch. This delay can be removed or changed at the expense of efficiency with `linger_ms`, `min_queued_messages`, and other keyword arguments (see readthedocs). You can still obtain delivery confirmation for messages, through a queue interface which can be enabled by setting `delivery_reports=True`. Here's a rough usage example:

```
>>> with topic.get_producer(delivery_reports=True) as producer:
...     count = 0
...     while True:
...         count += 1
...         producer.produce('test msg', partition_key='{}'.format(count))
...         if count % 10 ** 5 == 0:  # adjust this or bring lots of RAM ;)
...             while True:
...                 try:
...                     msg, exc = producer.get_delivery_report(block=False)
...                     if exc is not None:
...                         print 'Failed to deliver msg {}: {}'.format(
...                             msg.partition_key, repr(exc))
...                     else:
...                         print 'Successfully delivered msg {}'.format(
...                         msg.partition_key)
...                 except Queue.Empty:
...                     break
```

Note that the delivery report queue is thread-local: it will only serve reports for messages which were produced from the current thread. Also, if you're using *delivery_reports=True*, failing to consume the delivery report queue will cause PyKafka's memory usage to grow unbounded.

You can also consume messages from this topic using a *Consumer* instance.

```
>>> consumer = topic.get_simple_consumer()
>>> for message in consumer:
...     if message is not None:
...         print message.offset, message.value
0 test message 0
1 test message 1
2 test message 4
3 test message 9
```

This *SimpleConsumer* doesn't scale - if you have two *SimpleConsumers* consuming the same topic, they will receive duplicate messages. To get around this, you can use the *BalancedConsumer*.

```
>>> balanced_consumer = topic.get_balanced_consumer(
...     consumer_group='testgroup',
...     auto_commit_enable=True,
...     zookeeper_connect='myZkClusterNode1.com:2181,myZkClusterNode2.com:2181/
↪myZkChroot'
... )
```

You can have as many *BalancedConsumer* instances consuming a topic as that topic has partitions. If they are all connected to the same zookeeper instance, they will communicate with it to automatically balance the partitions between themselves. The partition assignment strategy used by the *BalancedConsumer* is the "range" strategy by default. The strategy is switchable via the *membership_protocol* keyword argument, and can be either an object exposed by *pykafka.membershipprotocol* or a custom instance of *pykafka.membershipprotocol.GroupMembershipProtocol*.

You can also use the Kafka 0.9 Group Membership API with the `managed` keyword argument on `get_balanced_consumer`.

CHAPTER 2

Using the librdkafka extension

PyKafka includes a C extension that makes use of librdkafka to speed up producer and consumer operation. To use the librdkafka extension, you need to make sure the header files and shared library are somewhere where python can find them, both when you build the extension (which is taken care of by `setup.py develop`) and at run time. Typically, this means that you need to either install librdkafka in a place conventional for your system, or declare `C_INCLUDE_PATH`, `LIBRARY_PATH`, and `LD_LIBRARY_PATH` in your shell environment to point to the installation location of the librdkafka shared objects. You can find this location with *locate librdkafka.so*.

After that, all that's needed is that you pass an extra parameter `use_rdkafka=True` to `topic.get_producer()`, `topic.get_simple_consumer()`, or `topic.get_balanced_consumer()`. Note that some configuration options may have different optimal values; it may be worthwhile to consult librdkafka's configuration notes for this.

CHAPTER 3

Operational Tools

PyKafka includes a small collection of CLI tools that can help with common tasks related to the administration of a Kafka cluster, including offset and lag monitoring and topic inspection. The full, up-to-date interface for these tools can be fould by running

```
$ python cli/kafka_tools.py --help
```

or after installing PyKafka via setuptools or pip:

```
$ kafka-tools --help
```

CHAPTER 4

PyKafka or kafka-python?

These are two different projects. See the discussion here for comparisons between the two projects.

CHAPTER 5

Contributing

If you're interested in contributing code to PyKafka, a good place to start is the "help wanted" issue tag. We also recommend taking a look at the contribution guide.

CHAPTER 6

Support

If you need help using PyKafka, there are a bunch of resources available. For usage questions or common recipes, check out the StackOverflow tag. The Google Group can be useful for more in-depth questions or inquries you'd like to send directly to the PyKafka maintainers. If you believe you've found a bug in PyKafka, please open a github issue after reading the contribution guide.

6.1 Help Documents

6.1.1 PyKafka Usage Guide

This document contains explanations and examples of common patterns of PyKafka usage.

6.1.2 Consumer Patterns

Setting the initial offset

When a PyKafka consumer starts fetching messages from a topic, its starting position in the log is defined by two keyword arguments: *auto_offset_reset* and *reset_offset_on_start*.

```
consumer = topic.get_simple_consumer(
    consumer_group="mygroup",
    auto_offset_reset=OffsetType.EARLIEST,
    reset_offset_on_start=False
)
```

The starting offset is also affected by whether or not the Kafka cluster holds any previously committed offsets for each consumer group/topic/partition set. In this document, a "new" group/topic/partition set is one for which Kafka does not hold any previously committed offsets, and an "existing" set is one for which Kafka does.

The consumer's initial behavior can be summed up by these rules:

- For any *new* group/topic/partitions, message consumption will start from *auto_offset_reset*. This is true independent of the value of *reset_offset_on_start*.
- For any *existing* group/topic/partitions, assuming *reset_offset_on_start=False*, consumption will start from the offset immediately following the last committed offset (if the last committed offset was 4, consumption starts at 5). If *reset_offset_on_start=True*, consumption starts from *auto_offset_reset*. If there is no committed offset, the group/topic/partition is considered *new*.

Put another way: if *reset_offset_on_start=True*, consumption will start from *auto_offset_reset*. If it is *False*, where consumption starts is dependent on the existence of committed offsets for the group/topic/partition in question.

Examples:

```
# assuming "mygroup" has no committed offsets

# starts from the latest available offset
consumer = topic.get_simple_consumer(
    consumer_group="mygroup",
    auto_offset_reset=OffsetType.LATEST
)
consumer.consume()
consumer.commit_offsets()

# starts from the last committed offset
consumer_2 = topic.get_simple_consumer(
    consumer_group="mygroup"
)

# starts from the earliest available offset
consumer_3 = topic.get_simple_consumer(
    consumer_group="mygroup",
    auto_offset_reset=OffsetType.EARLIEST,
    reset_offset_on_start=True
)
```

This behavior is based on the *auto.offset.reset* section of the Kafka documentation.

Consuming the last N messages from a topic

When you want to see only the last few messages of a topic, you can use the following pattern.

```
from __future__ import division

import math
from itertools import islice

from pykafka import KafkaClient
from pykafka.common import OffsetType

client = KafkaClient()
topic = client.topics['mytopic']
consumer = topic.get_simple_consumer(
    auto_offset_reset=OffsetType.LATEST,
    reset_offset_on_start=True)
LAST_N_MESSAGES = 50
# how many messages should we get from the end of each partition?
MAX_PARTITION_REWIND = int(math.ceil(LAST_N_MESSAGES / len(consumer._partitions)))
```

(continues on next page)

(continued from previous page)

```
# find the beginning of the range we care about for each partition
offsets = [(p, op.last_offset_consumed - MAX_PARTITION_REWIND)
           for p, op in consumer._partitions.iteritems()]
# if we want to rewind before the beginning of the partition, limit to beginning
offsets = [(p, (o if o > -1 else -2)) for p, o in offsets]
# reset the consumer's offsets
consumer.reset_offsets(offsets)
for message in islice(consumer, LAST_N_MESSAGES):
    print(message.offset, message.value)
```

op.last_offset_consumed is the "head" pointer of the consumer instance. Since we start by setting this consumer to *LATEST*, *last_offset_consumed* is the latest offset for the partition. Thus, *last_offset_consumed* - *MAX_PARTITION_REWIND* gives the starting offset of the last messages per partition.

6.1.3 Producer Patterns

Producing to multiple topics

Avoid repeated calls to the relatively *get_producer* when possible. If producing to multiple topics from a single process, it's helpful to keep the *Producer* objects in memory instead of letting them be garbage collected and reinstantiated repeatedly.

```
topic_producers = {topic.name: topic.get_producer() for topic in topics_to_produce_to}
for destination_topic, message in consumed_messages:
    topic_producers[destination_topic.name].produce(message)
```

6.1.4 Handling connection loss

The pykafka components are designed to raise exceptions when sufficient connection to the Kafka cluster cannot be established. There are cases in which some but not all of the brokers in a cluster are accessible to pykafka. In these cases, the component will attempt to continue operating. When it can't, an exception will be raised. Often this exception will be either *NoBrokersAvailableError* or *SocketDisconnectedError*. These exceptions should be caught and the component instance should be reinstantiated. In some cases, calling *stop(); start()* in response to these exceptions can be enough to establish a working connection.

```
from pykafka.exceptions import SocketDisconnectedError, NoBrokersAvailableError
# this illustrates consumer error catching; a similar method can be used for producers
consumer = topic.get_simple_consumer()
try:
    consumer.consume()
except (SocketDisconnectedError, NoBrokersAvailableError) as e:
    consumer = topic.get_simple_consumer()
    # use either the above method or the following:
    consumer.stop()
    consumer.start()
```

6.2 API Documentation

Note: PyKafka uses the convention that all class attributes prefixed with an underscore are considered private. They are not a part of the public interface, and thus are subject to change without a major version increment at any time.

Class attributes not prefixed with an underscore are treated as a fixed public API and are only changed in major version increments.

6.2.1 pykafka.balancedconsumer

class pykafka.balancedconsumer.**BalancedConsumer**(*topic*, *cluster*, *consumer_group*, *fetch_message_max_bytes=1048576*, *num_consumer_fetchers=1*, *auto_commit_enable=False*, *auto_commit_interval_ms=60000*, *queued_max_messages=2000*, *fetch_min_bytes=1*, *fetch_error_backoff_ms=500*, *fetch_wait_max_ms=100*, *offsets_channel_backoff_ms=1000*, *offsets_commit_max_retries=5*, *auto_offset_reset=-2*, *consumer_timeout_ms=-1*, *rebalance_max_retries=5*, *rebalance_backoff_ms=2000*, *zookeeper_connection_timeout_ms=6000*, *zookeeper_connect=None*, *zookeeper_hosts='127.0.0.1:2181'*, *zookeeper=None*, *auto_start=True*, *reset_offset_on_start=False*, *post_rebalance_callback=None*, *use_rdkafka=False*, *compacted_topic=False*, *membership_protocol=GroupMembershipProtocol(protocol_type='con*
protocol_name='range', *metadata=<pykafka.protocol.group_membership.ConsumerGroupP*
object>, *decide_partitions=<function decide_partitions_range>)*, *deserializer=None*, *reset_offset_on_fetch=True*)

Bases: object

A self-balancing consumer for Kafka that uses ZooKeeper to communicate with other balancing consumers.

Maintains a single instance of SimpleConsumer, periodically using the consumer rebalancing algorithm to reassign partitions to this SimpleConsumer.

__init__(*topic*, *cluster*, *consumer_group*, *fetch_message_max_bytes=1048576*, *num_consumer_fetchers=1*, *auto_commit_enable=False*, *auto_commit_interval_ms=60000*, *queued_max_messages=2000*, *fetch_min_bytes=1*, *fetch_error_backoff_ms=500*, *fetch_wait_max_ms=100*, *offsets_channel_backoff_ms=1000*, *offsets_commit_max_retries=5*, *auto_offset_reset=-2*, *consumer_timeout_ms=-1*, *rebalance_max_retries=5*, *rebalance_backoff_ms=2000*, *zookeeper_connection_timeout_ms=6000*, *zookeeper_connect=None*, *zookeeper_hosts='127.0.0.1:2181'*, *zookeeper=None*, *auto_start=True*, *reset_offset_on_start=False*, *post_rebalance_callback=None*, *use_rdkafka=False*, *compacted_topic=False*, *membership_protocol=GroupMembershipProtocol(protocol_type='consumer'*, *protocol_name='range'*, *metadata=<pykafka.protocol.group_membership.ConsumerGroupProtocolMetadata object>*, *decide_partitions=<function decide_partitions_range>)*, *deserializer=None*, *reset_offset_on_fetch=True*)

Create a BalancedConsumer instance

Parameters

- **topic** (*pykafka.topic.Topic*) – The topic this consumer should consume
- **cluster** (*pykafka.cluster.Cluster*) – The cluster to which this consumer should connect
- **consumer_group** (*str*) – The name of the consumer group this consumer should join. Consumer group names are namespaced at the cluster level, meaning that two consumers consuming different topics with the same group name will be treated as part of the same group.
- **fetch_message_max_bytes** (*int*) – The number of bytes of messages to attempt to fetch with each fetch request
- **num_consumer_fetchers** (*int*) – The number of workers used to make FetchRequests
- **auto_commit_enable** (*bool*) – If true, periodically commit to kafka the offset of messages already returned from consume() calls. Requires that *consumer_group* is not *None*.
- **auto_commit_interval_ms** (*int*) – The frequency (in milliseconds) at which the consumer's offsets are committed to kafka. This setting is ignored if *auto_commit_enable* is *False*.
- **queued_max_messages** (*int*) – The maximum number of messages buffered for consumption in the internal *pykafka.simpleconsumer.SimpleConsumer*
- **fetch_min_bytes** (*int*) – The minimum amount of data (in bytes) that the server should return for a fetch request. If insufficient data is available, the request will block until sufficient data is available.
- **fetch_error_backoff_ms** (*int*) – *UNUSED*. See *pykafka.simpleconsumer.SimpleConsumer*.
- **fetch_wait_max_ms** (*int*) – The maximum amount of time (in milliseconds) that the server will block before answering a fetch request if there isn't sufficient data to immediately satisfy *fetch_min_bytes*.
- **offsets_channel_backoff_ms** (*int*) – Backoff time to retry failed offset commits and fetches.
- **offsets_commit_max_retries** (*int*) – The number of times the offset commit worker should retry before raising an error.
- **auto_offset_reset** (*pykafka.common.OffsetType*) – What to do if an offset is out of range. This setting indicates how to reset the consumer's internal offset counter when an *OffsetOutOfRangeError* is encountered.
- **consumer_timeout_ms** (*int*) – Amount of time (in milliseconds) the consumer may spend without messages available for consumption before returning None.
- **rebalance_max_retries** (*int*) – The number of times the rebalance should retry before raising an error.
- **rebalance_backoff_ms** (*int*) – Backoff time (in milliseconds) between retries during rebalance.
- **zookeeper_connection_timeout_ms** (*int*) – The maximum time (in milliseconds) that the consumer waits while establishing a connection to zookeeper.

- **zookeeper_connect** (*str*) – Deprecated::2.7,3.6 Comma-Separated (ip1:port1,ip2:port2) strings indicating the zookeeper nodes to which to connect.
- **zookeeper_hosts** (*str*) – KazooClient-formatted string of ZooKeeper hosts to which to connect.
- **zookeeper** (kazoo.client.KazooClient) – A KazooClient connected to a Zookeeper instance. If provided, *zookeeper_connect* is ignored.
- **auto_start** (*bool*) – Whether the consumer should begin communicating with zookeeper after __init__ is complete. If false, communication can be started with *start()*.
- **reset_offset_on_start** (*bool*) – Whether the consumer should reset its internal offset counter to *self._auto_offset_reset* and commit that offset immediately upon starting up
- **post_rebalance_callback** (*function*) – A function to be called when a rebalance is in progress. This function should accept three arguments: the *pykafka.balancedconsumer.BalancedConsumer* instance that just completed its rebalance, a dict of partitions that it owned before the rebalance, and a dict of partitions it owns after the rebalance. These dicts map partition ids to the most recently known offsets for those partitions. This function can optionally return a dictionary mapping partition ids to offsets. If it does, the consumer will reset its offsets to the supplied values before continuing consumption. Note that the BalancedConsumer is in a poorly defined state at the time this callback runs, so that accessing its properties (such as *held_offsets* or *partitions*) might yield confusing results. Instead, the callback should really rely on the provided partition-id dicts, which are well-defined.
- **use_rdkafka** (*bool*) – Use librdkafka-backed consumer if available
- **compacted_topic** (*bool*) – Set to read from a compacted topic. Forces consumer to use less stringent message ordering logic because compacted topics do not provide offsets in strict incrementing order.
- **membership_protocol** (*pykafka.membershipprotocol.GroupMembershipProtocol*) – The group membership protocol to which this consumer should adhere
- **deserializer** (*function*) – A function defining how to deserialize messages returned from Kafka. A function with the signature d(value, partition_key) that returns a tuple of (deserialized_value, deserialized_partition_key). The arguments passed to this function are the bytes representations of a message's value and partition key, and the returned data should be these fields transformed according to the client code's serialization logic. See *pykafka.utils.__init__* for stock implemtations.
- **reset_offset_on_fetch** (*bool*) – Whether to update offsets during fetch_offsets. Disable for read-only use cases to prevent side-effects.

__iter__()
Yield an infinite stream of messages until the consumer times out

__repr__(*)* <==> *repr(x)*

__weakref__
list of weak references to the object (if defined)

_add_partitions(*partitions*)
Add partitions to the zookeeper registry for this consumer.

Parameters **partitions** (Iterable of *pykafka.partition.Partition*) – The partitions to add.

_add_self()
Register this consumer in zookeeper.

_build_watch_callback(*fn*, *proxy*)
Return a function that's safe to use as a ChildrenWatch callback

Fixes the issue from https://github.com/Parsely/pykafka/issues/345

_get_held_partitions()
Build a set of partitions zookeeper says we own

_get_internal_consumer(*partitions=None*, *start=True*)
Instantiate a SimpleConsumer for internal use.

If there is already a SimpleConsumer instance held by this object, disable its workers and mark it for garbage collection before creating a new one.

_get_participants()
Use zookeeper to get the other consumers of this topic.

Returns A sorted list of the ids of other consumers of this consumer's topic

_partitions
Convenient shorthand for set of partitions internally held

_path_from_partition(*p*)
Given a partition, return its path in zookeeper.

_path_self
Path where this consumer should be registered in zookeeper

_raise_worker_exceptions()
Raises exceptions encountered on worker threads

_rebalance()
Start the rebalancing process for this consumer

This method is called whenever a zookeeper watch is triggered.

_remove_partitions(*partitions*)
Remove partitions from the zookeeper registry for this consumer.

Parameters **partitions** (Iterable of *pykafka.partition.Partition*) – The partitions to remove.

_set_watches()
Set watches in zookeeper that will trigger rebalances.

Rebalances should be triggered whenever a broker, topic, or consumer znode is changed in zookeeper. This ensures that the balance of the consumer group remains up-to-date with the current state of the cluster.

_setup_internal_consumer(*partitions=None*, *start=True*)
Instantiate an internal SimpleConsumer instance

_setup_zookeeper(*zookeeper_connect*, *timeout*)
Open a connection to a ZooKeeper host.

Parameters

- **zookeeper_connect** (*str*) – The 'ip:port' address of the zookeeper node to which to connect.
- **timeout** (*int*) – Connection timeout (in milliseconds)

_update_member_assignment()
Decide and assign new partitions for this consumer

commit_offsets(*partition_offsets=None*)
Commit offsets for this consumer's partitions

Uses the offset commit/fetch API

Parameters **partition_offsets** (Sequence of tuples of the form (*pykafka.partition.Partition*, int)) – (*partition*, *offset*) pairs to commit where *partition* is the partition for which to commit the offset and *offset* is the offset to commit for the partition. Note that using this argument when *auto_commit_enable* is enabled can cause inconsistencies in committed offsets. For best results, use *either* this argument *or* *auto_commit_enable*.

consume(*block=True*)
Get one message from the consumer

Parameters **block** (*bool*) – Whether to block while waiting for a message

held_offsets
Return a map from partition id to held offset for each partition

partitions
A list of the partitions that this consumer consumes

reset_offsets(*partition_offsets=None*)
Reset offsets for the specified partitions

For each value provided in *partition_offsets*: if the value is an integer, immediately reset the partition's internal offset counter to that value. If it's a *datetime.datetime* instance or a valid *OffsetType*, issue a *ListOffsetRequest* using that timestamp value to discover the latest offset in the latest log segment before that timestamp, then set the partition's internal counter to that value.

Parameters **partition_offsets** (Sequence of tuples of the form (*pykafka.partition.Partition*, int OR *datetime.datetime*)) – (*partition*, *timestamp_or_offset*) pairs to reset where *partition* is the partition for which to reset the offset and *timestamp_or_offset* is EITHER the timestamp before which to find a valid offset to set the partition's counter to OR the new offset the partition's counter should be set to

start()
Open connections and join a consumer group.

stop()
Close the zookeeper connection and stop consuming.

This method should be called as part of a graceful shutdown process.

topic
The topic this consumer consumes

6.2.2 pykafka.broker

Author: Keith Bourgoin, Emmett Butler

class pykafka.broker.**Broker**(*id_*, *host*, *port*, *handler*, *socket_timeout_ms*, *offsets_channel_socket_timeout_ms*, *buffer_size=1048576*, *source_host=''*, *source_port=0*, *ssl_config=None*, *broker_version='0.9.0'*, *api_versions=None*)
Bases: object

A Broker is an abstraction over a real kafka server instance. It is used to perform requests to these servers.

__init__(*id_*, *host*, *port*, *handler*, *socket_timeout_ms*, *offsets_channel_socket_timeout_ms*, *buffer_size=1048576*, *source_host=''*, *source_port=0*, *ssl_config=None*, *broker_version='0.9.0'*, *api_versions=None*)

Create a Broker instance.

Parameters

- **id** (*int*) – The id number of this broker
- **host** (*str*) – The host address to which to connect. An IP address or a DNS name
- **port** (*int*) – The port on which to connect
- **handler** (*pykafka.handlers.Handler*) – A Handler instance that will be used to service requests and responses
- **socket_timeout_ms** (*int*) – The socket timeout for network requests
- **offsets_channel_socket_timeout_ms** (*int*) – The socket timeout for network requests on the offsets channel
- **buffer_size** (*int*) – The size (bytes) of the internal buffer used to receive network responses
- **source_host** (*str*) – The host portion of the source address for socket connections
- **source_port** (*int*) – The port portion of the source address for socket connections
- **ssl_config** (*pykafka.connection.SslConfig*) – Config object for SSL connection
- **broker_version** (*str*) – The protocol version of the cluster being connected to. If this parameter doesn't match the actual broker version, some pykafka features may not work properly.
- **api_versions** (Iterable of `pykafka.protocol.ApiVersionsSpec`) – A sequence of `pykafka.protocol.ApiVersionsSpec` objects indicating the API version compatibility of this broker

__repr__(*)* <==> *repr(x)*

__weakref__

list of weak references to the object (if defined)

_get_unique_req_handler(*connection_id*)

Return a RequestHandler instance unique to the given connection_id

In some applications, for example the Group Membership API, requests running in the same process must be interleaved. When both of these requests are using the same RequestHandler instance, the requests are queued and the interleaving semantics are not upheld. This method behaves identically to self._req_handler if there is only one connection_id per KafkaClient. If a single KafkaClient needs to use more than one connection_id, this method maintains a dictionary of connections unique to those ids.

Parameters **connection_id** (*str*) – The unique identifier of the connection to return

commit_consumer_group_offsets(*consumer_group*, *consumer_group_generation_id*, *consumer_id*, *preqs*)

Commit offsets to Kafka using the Offset Commit/Fetch API

Commit the offsets of all messages consumed so far by this consumer group with the Offset Commit/Fetch API

Based on Step 2 here https://cwiki.apache.org/confluence/display/KAFKA/Committing+and+fetching+consumer+offsets+in+Kafka

Parameters

- **consumer_group** (*str*) – the name of the consumer group for which to commit offsets
- **consumer_group_generation_id** (*int*) – The generation ID for this consumer group
- **consumer_id** (*str*) – The identifier for this consumer group
- **preqs** (Iterable of *pykafka.protocol.PartitionOffsetCommitRequest*) – Requests indicating the partitions for which offsets should be committed

connect (*attempts=3*)

Establish a connection to the broker server.

Creates a new *pykafka.connection.BrokerConnection* and a new *pykafka.handlers.RequestHandler* for this broker

connect_offsets_channel (*attempts=3*)

Establish a connection to the Broker for the offsets channel

Creates a new *pykafka.connection.BrokerConnection* and a new *pykafka.handlers.RequestHandler* for this broker's offsets channel

connected

Returns True if this object's main connection to the Kafka broker is active

fetch_consumer_group_offsets (*consumer_group*, *preqs*)

Fetch the offsets stored in Kafka with the Offset Commit/Fetch API

Based on Step 2 here https://cwiki.apache.org/confluence/display/KAFKA/Committing+and+fetching+consumer+offsets+in+Kafka

Parameters

- **consumer_group** (*str*) – the name of the consumer group for which to fetch offsets
- **preqs** (Iterable of *pykafka.protocol.PartitionOffsetFetchRequest*) – Requests indicating the partitions for which offsets should be fetched

classmethod from_metadata (*metadata*, *handler*, *socket_timeout_ms*, *offsets_channel_socket_timeout_ms*, *buffer_size=65536*, *source_host=''*, *source_port=0*, *ssl_config=None*, *broker_version='0.9.0'*, *api_versions=None*)

Create a Broker using BrokerMetadata

Parameters

- **metadata** (pykafka.protocol.BrokerMetadata.) – Metadata that describes the broker.
- **handler** (*pykafka.handlers.Handler*) – A Handler instance that will be used to service requests and responses
- **socket_timeout_ms** (*int*) – The socket timeout for network requests
- **offsets_channel_socket_timeout_ms** (*int*) – The socket timeout for network requests on the offsets channel
- **buffer_size** (*int*) – The size (bytes) of the internal buffer used to receive network responses
- **source_host** (*str*) – The host portion of the source address for socket connections

- **source_port** (*int*) – The port portion of the source address for socket connections
- **ssl_config** (*pykafka.connection.SslConfig*) – Config object for SSL connection
- **broker_version** (*str*) – The protocol version of the cluster being connected to. If this parameter doesn't match the actual broker version, some pykafka features may not work properly.
- **api_versions** (Iterable of `pykafka.protocol.ApiVersionsSpec`) – A sequence of `pykafka.protocol.ApiVersionsSpec` objects indicating the API version compatibility of this broker

handler
The primary *pykafka.handlers.RequestHandler* for this broker

This handler handles all requests outside of the commit/fetch api

heartbeat(*connection_id*, *consumer_group*, *generation_id*, *member_id*)
Send a HeartbeatRequest

Parameters

- **connection_id** (*str*) – The unique identifier of the connection on which to make this request
- **consumer_group** (*bytes*) – The name of the consumer group to which this consumer belongs
- **generation_id** (*int*) – The current generation for the consumer group
- **member_id** (*bytes*) – The ID of the consumer sending this heartbeat

host
The host to which this broker is connected

id
The broker's ID within the Kafka cluster

join_group(*connection_id*, *consumer_group*, *member_id*, *topic_name*, *membership_protocol*)
Send a JoinGroupRequest

Parameters

- **connection_id** (*str*) – The unique identifier of the connection on which to make this request
- **consumer_group** (*bytes*) – The name of the consumer group to join
- **member_id** (*bytes*) – The ID of the consumer joining the group
- **topic_name** (*str*) – The name of the topic to which to connect, used in protocol metadata
- **membership_protocol** (*pykafka.membershipprotocol.GroupMembershipProtocol*) – The group membership protocol to which this request should adhere

leave_group(*connection_id*, *consumer_group*, *member_id*)
Send a LeaveGroupRequest

Parameters

- **connection_id** (*str*) – The unique identifier of the connection on which to make this request

- **consumer_group** (*bytes*) – The name of the consumer group to leave
- **member_id** (*bytes*) – The ID of the consumer leaving the group

offsets_channel_connected
Returns True if this object's offsets channel connection to the Kafka broker is active

offsets_channel_handler

The offset channel *pykafka.handlers.RequestHandler* for this broker

This handler handles all requests that use the commit/fetch api

port
The port where the broker is available

sync_group(*connection_id*, *consumer_group*, *generation_id*, *member_id*, *group_assignment*)
Send a SyncGroupRequest

Parameters

- **connection_id** (*str*) – The unique identifier of the connection on which to make this request
- **consumer_group** (*bytes*) – The name of the consumer group to which this consumer belongs
- **generation_id** (*int*) – The current generation for the consumer group
- **member_id** (*bytes*) – The ID of the consumer syncing
- **group_assignment** (iterable of *pykafka.protocol.MemberAssignment*) – A sequence of *pykafka.protocol.MemberAssignment* instances indicating the partition assignments for each member of the group. When *sync_group* is called by a member other than the leader of the group, *group_assignment* should be an empty sequence.

6.2.3 pykafka.client

Author: Keith Bourgoin, Emmett Butler

class pykafka.client.**KafkaClient**(*hosts='127.0.0.1:9092'*, *zookeeper_hosts=None*, *socket_timeout_ms=30000*, *offsets_channel_socket_timeout_ms=10000*, *use_greenlets=False*, *exclude_internal_topics=True*, *source_address=''*, *ssl_config=None*, *broker_version='0.9.0'*)

Bases: object

A high-level pythonic client for Kafka

NOTE: *KafkaClient* holds weak references to *Topic* instances via pykafka.cluster.TopicDict. To perform operations directly on these topics, such as examining their partition lists, client code must hold a strong reference to the topics it cares about. If client code doesn't need to examine *Topic* instances directly, no strong references are necessary.

Notes on Zookeeper: Zookeeper is used by kafka and its clients to store several types of information, including broker host strings, partition ownerships, and depending on your kafka version, consumer offsets. The kafka-console-* tools rely on zookeeper to discover brokers - this is why you can't directly specify a broker to these tools and are required to give a zookeeper host string. In theory, this insulates you as a user of the console tools from having to care about which specific brokers in your kafka cluster might be accessible at any given time.

In pykafka, the paradigm is slightly different, though the above method is also supported. When you instantiate a *KafkaClient*, you can specify either *hosts* or *zookeeper_hosts*. *hosts* is a comma-separated list of brokers to which to connect, and *zookeeper_hosts* is a zookeeper connection string. If you specify *zookeeper_hosts*, it overrides *hosts*. Thus you can create a *KafkaClient* that is connected to your kafka cluster by providing either a zookeeper or a broker connection string.

As for why the specific components do and don't require knowledge of the zookeeper cluster, there are some different reasons. *SimpleConsumer*, since it does not perform consumption balancing, does not actually require access to zookeeper at all. Since kafka 0.8.2, consumer offset information is stored by the kafka broker itself instead of the zookeeper cluster. The *BalancedConsumer*, by contrast, requires explicit knowledge of the zookeeper cluster because it performs consumption balancing. Zookeeper stores the information about which consumers own which partitions and provides a central repository of that information for all consumers to read. The *BalancedConsumer* cannot do what it does without direct access to zookeeper for this reason. Note that the *ManagedBalancedConsumer*, which works with kafka 0.9 and above, removes this dependency on zookeeper from the balanced consumption process by storing partition ownership information in the kafka broker.

The *Producer* is allowed to send messages to whatever partitions it wants. In pykafka, by default the partition for each message is chosen randomly to provide an even distribution of messages across partitions. The producer actually doesn't do anything that requires information stored in zookeeper, and since the connection to the kafka cluster is handled by the above-mentioned logic in *KafkaClient*, it doesn't need the zookeeper host string at all.

__init__ (*hosts='127.0.0.1:9092'*, *zookeeper_hosts=None*, *socket_timeout_ms=30000*, *offsets_channel_socket_timeout_ms=10000*, *use_greenlets=False*, *exclude_internal_topics=True*, *source_address=''*, *ssl_config=None*, *broker_version='0.9.0'*)

Create a connection to a Kafka cluster.

Documentation for source_address can be found at https://docs.python.org/2/library/socket.html#socket.create_connection

Parameters

- **hosts** (*str*) – Comma-separated list of kafka hosts to which to connect. If *ssl_config* is specified, the ports specified here are assumed to be SSL ports
- **zookeeper_hosts** (*str*) – KazooClient-formatted string of ZooKeeper hosts to which to connect. If not *None*, this argument takes precedence over *hosts*
- **socket_timeout_ms** (*int*) – The socket timeout (in milliseconds) for network requests
- **offsets_channel_socket_timeout_ms** (*int*) – The socket timeout (in milliseconds) when reading responses for offset commit and offset fetch requests.
- **use_greenlets** (*bool*) – Whether to perform parallel operations on greenlets instead of OS threads
- **exclude_internal_topics** (*bool*) – Whether messages from internal topics (specifically, the offsets topic) should be exposed to the consumer.
- **source_address** (str *'host:port'*) – The source address for socket connections
- **ssl_config** (*pykafka.connection.SslConfig*) – Config object for SSL connection
- **broker_version** (*str*) – The protocol version of the cluster being connected to. If this parameter doesn't match the actual broker version, some pykafka features may not work properly.

__repr__ (*)* <==> *repr(x)*

__weakref__

list of weak references to the object (if defined)

update_cluster()
Update known brokers and topics.

Updates each Topic and Broker, adding new ones as found, with current metadata from the cluster.

6.2.4 pykafka.cluster

class pykafka.cluster.**Cluster**(*hosts*, *handler*, *socket_timeout_ms=30000*, *offsets_channel_socket_timeout_ms=10000*, *exclude_internal_topics=True*, *source_address=''*, *zookeeper_hosts=None*, *ssl_config=None*, *broker_version='0.9.0'*)

Bases: object

A Cluster is a high-level abstraction of the collection of brokers and topics that makes up a real kafka cluster.

__init__(*hosts*, *handler*, *socket_timeout_ms=30000*, *offsets_channel_socket_timeout_ms=10000*, *exclude_internal_topics=True*, *source_address=''*, *zookeeper_hosts=None*, *ssl_config=None*, *broker_version='0.9.0'*)
Create a new Cluster instance.

Parameters

- **hosts** (*str*) – Comma-separated list of kafka hosts to which to connect.
- **zookeeper_hosts** (*str*) – KazooClient-formatted string of ZooKeeper hosts to which to connect. If not *None*, this argument takes precedence over *hosts*
- **handler** (*pykafka.handlers.Handler*) – The concurrency handler for network requests.
- **socket_timeout_ms** (*int*) – The socket timeout (in milliseconds) for network requests
- **offsets_channel_socket_timeout_ms** (*int*) – The socket timeout (in milliseconds) when reading responses for offset commit and offset fetch requests.
- **exclude_internal_topics** (*bool*) – Whether messages from internal topics (specifically, the offsets topic) should be exposed to consumers.
- **source_address** (str *'host:port'*) – The source address for socket connections
- **ssl_config** (*pykafka.connection.SslConfig*) – Config object for SSL connection
- **broker_version** (*str*) – The protocol version of the cluster being connected to. If this parameter doesn't match the actual broker version, some pykafka features may not work properly.

__repr__() <==> *repr(x)*

__weakref__
list of weak references to the object (if defined)

_get_broker_connection_info()
Get a list of host:port pairs representing possible broker connections

For use only when self.brokers is not populated (ie at startup)

_get_brokers_from_zookeeper(*zk_connect*)
Build a list of broker connection pairs from a ZooKeeper host

Parameters **zk_connect** (*str*) – The ZooKeeper connect string of the instance to which to connect

_get_metadata(*topics=None*)
Get fresh cluster metadata from a broker.

_request_random_broker(*broker_connects*, *req_fn*)
Make a request to any broker in broker_connects

Returns the result of the first successful request

Parameters

- **broker_connects** (*Iterable of two-element sequences of the format (broker_host, broker_port)*) – The set of brokers to which to attempt to connect
- **req_fn** (*function*) – A function accepting a *pykafka.broker.Broker* as its sole argument that returns a `pykafka.protocol.Response`. The argument to this function will be the each of the brokers discoverable via *broker_connects* in turn.

_update_brokers(*broker_metadata*, *controller_id*)
Update brokers with fresh metadata.

Parameters

- **broker_metadata** (Dict of *{name: metadata}* where *metadata* is `pykafka.protocol.BrokerMetadata` and *name* is str.) – Metadata for all brokers.
- **controller_id** (*int*) – The ID of the cluster's controller broker, if applicable

brokers
The dict of known brokers for this cluster

fetch_api_versions()
Get API version info from an available broker and save it

get_group_coordinator(*consumer_group*)
Get the broker designated as the group coordinator for this consumer group.

Based on Step 1 at https://cwiki.apache.org/confluence/display/KAFKA/Committing+and+fetching+consumer+offsets+in+Kafka

Parameters **consumer_group** (*str*) – The name of the consumer group for which to find the offset manager.

get_managed_group_descriptions()
Return detailed descriptions of all managed consumer groups on this cluster

This function only returns descriptions for consumer groups created via the Group Management API, which pykafka refers to as :class:'ManagedBalancedConsumer's

handler
The concurrency handler for network requests

topics
The dict of known topics for this cluster

NOTE: This dict is an instance of `pykafka.cluster.TopicDict`, which uses weak references and lazy evaluation to avoid instantiating unnecessary *pykafka.Topic* objects. Thus, the values displayed when printing *client.topics* on a freshly created `pykafka.KafkaClient` will be *None*. This simply means that the topic instances have not yet been created, but they will be when *__getitem__* is called on the dictionary.

update()
Update known brokers and topics.

6.2.5 pykafka.common

Author: Keith Bourgoin

class pykafka.common.**Message**
Bases: object

Message class.

Variables

- **response_code** – Response code from Kafka
- ***topic*** – Originating topic
- **payload** – Message payload
- **key** – (optional) Message key
- ***offset*** – Message offset

class pykafka.common.**CompressionType**
Bases: object

Enum for the various compressions supported.

Variables

- **NONE** – Indicates no compression in use
- **GZIP** – Indicates gzip compression in use
- **SNAPPY** – Indicates snappy compression in use

__weakref__
list of weak references to the object (if defined)

class pykafka.common.**OffsetType**
Bases: object

Enum for special values for earliest/latest offsets.

Variables

- **EARLIEST** – Indicates the earliest offset available for a partition
- **LATEST** – Indicates the latest offset available for a partition

__weakref__
list of weak references to the object (if defined)

6.2.6 pykafka.connection

class pykafka.connection.**SslConfig**(*cafile*, *certfile=None*, *keyfile=None*, *password=None*)
Bases: object

Config object for SSL connections

This aims to pick optimal defaults for the majority of use cases. If you have special requirements (eg. you want to enable hostname checking), you may monkey-patch *self._wrap_socket* (see *_legacy_wrap_socket()* for an example) before passing the *SslConfig* to *KafkaClient* init, like so:

config = SslConfig(cafile='/your/ca/file') config._wrap_socket = config._legacy_wrap_socket() client = KafkaClient('localhost:<ssl-port>', ssl_config=config)

Alternatively, completely supplanting this class with your own is also simple: if you are not going to be using the *pykafka.rdkafka* classes, only a method *wrap_socket()* is expected (so you can eg. simply pass in a plain *ssl.SSLContext* instance instead). The *pykafka.rdkafka* classes require four further attributes: *cafile*, *certfile*, *keyfile*, and *password* (the *SslConfig.__init__* docstring explains their meaning)

__init__(*cafile*, *certfile=None*, *keyfile=None*, *password=None*)
Specify certificates for SSL connection

Parameters

- **cafile** (*str*) – Path to trusted CA certificate
- **certfile** (*str*) – Path to client certificate
- **keyfile** (*str*) – Path to client private-key file
- **password** (*bytes*) – Password for private key

__weakref__
list of weak references to the object (if defined)

_legacy_wrap_socket()
Create socket-wrapper on a pre-2.7.9 Python interpreter

wrap_socket(*sock*)
Wrap a socket in an SSL context (see *ssl.wrap_socket*)

Parameters **socket** (socket.socket) – Plain socket

class pykafka.connection.**BrokerConnection**(*host*, *port*, *handler*, *buffer_size=1048576*, *source_host=''*, *source_port=0*, *ssl_config=None*)
Bases: object

BrokerConnection thinly wraps a *socket.create_connection* call and handles the sending and receiving of data that conform to the kafka binary protocol over that socket.

__del__()
Close this connection when the object is deleted.

__init__(*host*, *port*, *handler*, *buffer_size=1048576*, *source_host=''*, *source_port=0*, *ssl_config=None*)
Initialize a socket connection to Kafka.

Parameters

- **host** (*str*) – The host to which to connect
- **port** (*int*) – The port on the host to which to connect. Assumed to be an ssl-endpoint if (and only if) *ssl_config* is also provided
- **handler** (*pykafka.handlers.Handler*) – The *pykafka.handlers.Handler* instance to use when creating a connection
- **buffer_size** (*int*) – The size (in bytes) of the buffer in which to hold response data.
- **source_host** (*str*) – The host portion of the source address for the socket connection
- **source_port** (*int*) – The port portion of the source address for the socket connection
- **ssl_config** (*pykafka.connection.SslConfig*) – Config object for SSL connection

__weakref__
list of weak references to the object (if defined)

connect(*timeout*, *attempts=1*)
Connect to the broker, retrying if specified.

connected
Returns true if the socket connection is open.

disconnect()
Disconnect from the broker.

reconnect()
Disconnect from the broker, then reconnect

request(*request*)
Send a request over the socket connection

response()
Wait for a response from the broker

6.2.7 pykafka.exceptions

Author: Keith Bourgoin, Emmett Butler

exception pykafka.exceptions.**ConsumerStoppedException**
Bases: *pykafka.exceptions.KafkaException*

Indicates that the consumer was stopped when an operation was attempted that required it to be running

exception pykafka.exceptions.**GroupAuthorizationFailed**
Bases: *pykafka.exceptions.ProtocolClientError*

Returned by the broker when the client is not authorized to access a particular groupId.

exception pykafka.exceptions.**GroupCoordinatorNotAvailable**
Bases: *pykafka.exceptions.ProtocolClientError*

The broker returns this error code for consumer metadata requests or offset commit requests if the offsets topic has not yet been created.

exception pykafka.exceptions.**GroupLoadInProgress**
Bases: *pykafka.exceptions.ProtocolClientError*

The broker returns this error code for an offset fetch request if it is still loading offsets (after a leader change for that offsets topic partition), or in response to group membership requests (such as heartbeats) when group metadata is being loaded by the coordinator.

exception pykafka.exceptions.**IllegalGeneration**
Bases: *pykafka.exceptions.ProtocolClientError*

Returned from group membership requests (such as heartbeats) when the generation id provided in the request is not the current generation

exception pykafka.exceptions.**InconsistentGroupProtocol**
Bases: *pykafka.exceptions.ProtocolClientError*

Returned in join group when the member provides a protocol type or set of protocols which is not compatible with the current group.

exception pykafka.exceptions.**InvalidMessageError**
Bases: *pykafka.exceptions.ProtocolClientError*

This indicates that a message contents does not match its CRC

exception pykafka.exceptions.**InvalidMessageSize**
Bases: *pykafka.exceptions.ProtocolClientError*

The message has a negative size

exception pykafka.exceptions.**InvalidSessionTimeout**
Bases: *pykafka.exceptions.ProtocolClientError*

Returned in join group when the requested session timeout is outside of the allowed range on the broker

exception pykafka.exceptions.**InvalidTopic**
Bases: *pykafka.exceptions.ProtocolClientError*

For a request which attempts to access an invalid topic (e.g. one which has an illegal name), or if an attempt is made to write to an internal topic (such as the consumer offsets topic).

exception pykafka.exceptions.**KafkaException**
Bases: exceptions.Exception

Generic exception type. The base of all pykafka exception types.

__weakref__
list of weak references to the object (if defined)

exception pykafka.exceptions.**LeaderNotAvailable**
Bases: *pykafka.exceptions.ProtocolClientError*

This error is thrown if we are in the middle of a leadership election and there is currently no leader for this partition and hence it is unavailable for writes.

exception pykafka.exceptions.**LeaderNotFoundError**
Bases: *pykafka.exceptions.KafkaException*

Indicates that the leader broker for a given partition was not found during an update in response to a MetadataRequest

exception pykafka.exceptions.**MessageSetDecodeFailure**
Bases: *pykafka.exceptions.KafkaException*

Indicates a generic failure in the decoding of a MessageSet from the broker

exception pykafka.exceptions.**MessageSizeTooLarge**
Bases: *pykafka.exceptions.ProtocolClientError*

The server has a configurable maximum message size to avoid unbounded memory allocation. This error is thrown if the client attempts to produce a message larger than this maximum.

exception pykafka.exceptions.**NoBrokersAvailableError**
Bases: *pykafka.exceptions.KafkaException*

Indicates that no brokers were available to the cluster's metadata update attempts

exception pykafka.exceptions.**NoMessagesConsumedError**
Bases: *pykafka.exceptions.KafkaException*

Indicates that no messages were returned from a MessageSet

exception pykafka.exceptions.**NotCoordinatorForGroup**
Bases: *pykafka.exceptions.ProtocolClientError*

The broker returns this error code if it receives an offset fetch or commit request for a consumer group that it is not a coordinator for.

exception pykafka.exceptions.**NotLeaderForPartition**
Bases: *pykafka.exceptions.ProtocolClientError*

This error is thrown if the client attempts to send messages to a replica that is not the leader for some partition. It indicates that the client's metadata is out of date.

exception pykafka.exceptions.**OffsetMetadataTooLarge**
Bases: *pykafka.exceptions.ProtocolClientError*

If you specify a string larger than configured maximum for offset metadata

exception pykafka.exceptions.**OffsetOutOfRangeError**
Bases: *pykafka.exceptions.ProtocolClientError*

The requested offset is outside the range of offsets maintained by the server for the given topic/partition.

exception pykafka.exceptions.**OffsetRequestFailedError**
Bases: *pykafka.exceptions.KafkaException*

Indicates that OffsetRequests for offset resetting failed more times than the configured maximum

exception pykafka.exceptions.**PartitionOwnedError**(*partition*, **args*, ***kwargs*)
Bases: *pykafka.exceptions.KafkaException*

Indicates a given partition is still owned in Zookeeper.

__init__(*partition*, **args*, ***kwargs*)
x.__init__(...) initializes x; see help(type(x)) for signature

exception pykafka.exceptions.**ProduceFailureError**
Bases: *pykafka.exceptions.KafkaException*

Indicates a generic failure in the producer

exception pykafka.exceptions.**ProducerQueueFullError**
Bases: *pykafka.exceptions.KafkaException*

Indicates that one or more of the AsyncProducer's internal queues contain at least max_queued_messages messages

exception pykafka.exceptions.**ProducerStoppedException**
Bases: *pykafka.exceptions.KafkaException*

Raised when the Producer is used while not running

exception pykafka.exceptions.**ProtocolClientError**
Bases: *pykafka.exceptions.KafkaException*

Base class for protocol errors

exception pykafka.exceptions.**RdKafkaException**
Bases: *pykafka.exceptions.KafkaException*

Error in rdkafka extension that hasn't any equivalent pykafka exception

In *pykafka.rdkafka._rd_kafka* we try hard to emit the same exceptions that the pure pykafka classes emit. This is a fallback for the few cases where we can't find a suitable exception

exception pykafka.exceptions.**RdKafkaStoppedException**
Bases: *pykafka.exceptions.RdKafkaException*

Consumer or producer handle was stopped

Raised by the C extension, to be translated to ConsumerStoppedException or ProducerStoppedException by the caller

exception pykafka.exceptions.**RebalanceInProgress**
Bases: *pykafka.exceptions.ProtocolClientError*

Returned in heartbeat requests when the coordinator has begun rebalancing the group. This indicates to the client that it should rejoin the group.

exception pykafka.exceptions.**RequestTimedOut**
Bases: *pykafka.exceptions.ProtocolClientError*

This error is thrown if the request exceeds the user-specified time limit in the request.

exception pykafka.exceptions.**SocketDisconnectedError**
Bases: *pykafka.exceptions.KafkaException*

Indicates that the socket connecting this client to a kafka broker has become disconnected

exception pykafka.exceptions.**TopicAuthorizationFailed**
Bases: *pykafka.exceptions.ProtocolClientError*

Returned by the broker when the client is not authorized to access the requested topic.

exception pykafka.exceptions.**UnicodeException**
Bases: exceptions.Exception

Indicates that an error was encountered while processing a unicode string

__weakref__
list of weak references to the object (if defined)

exception pykafka.exceptions.**UnknownError**
Bases: *pykafka.exceptions.ProtocolClientError*

An unexpected server error

exception pykafka.exceptions.**UnknownMemberId**
Bases: *pykafka.exceptions.ProtocolClientError*

Returned from group requests (offset commits/fetches, heartbeats, etc) when the memberId is not in the current generation. Also returned if SimpleConsumer is incorrectly instantiated with a non-default consumer_id.

exception pykafka.exceptions.**UnknownTopicOrPartition**
Bases: *pykafka.exceptions.ProtocolClientError*

This request is for a topic or partition that does not exist on this broker.

6.2.8 pykafka.handlers

Author: Keith Bourgoin, Emmett Butler

class pykafka.handlers.**ResponseFuture**(*handler*)
Bases: object

A response which may have a value at some point.

__init__(*handler*)

__weakref__
list of weak references to the object (if defined)

get(*response_cls=None*, *timeout=None*, ***response_kwargs*)
Block until data is ready and return.

Raises an exception if there was an error.

set_error(*error*)
Set error and trigger get method.

set_response(*response*)
Set response data and trigger get method.

class pykafka.handlers.**Handler**
Bases: object

Base class for Handler classes

__weakref__
list of weak references to the object (if defined)

spawn(*target*, **args*, ***kwargs*)
Create the worker that will process the work to be handled

class pykafka.handlers.**ThreadingHandler**
Bases: *pykafka.handlers.Handler*

A handler that uses a threading.Thread to perform its work

Event(***kwargs*)
A factory function that returns a new event.

Events manage a flag that can be set to true with the set() method and reset to false with the clear() method. The wait() method blocks until the flag is true.

GaiError
alias of socket.gaierror

Lock()
allocate_lock() -> lock object (allocate() is an obsolete synonym)

Create a new lock object. See help(LockType) for information about locks.

class Queue(*maxsize=0*)
Create a queue object with a given maximum size.

If maxsize is <= 0, the queue size is infinite.

empty()
Return True if the queue is empty, False otherwise (not reliable!).

full()
Return True if the queue is full, False otherwise (not reliable!).

get(*block=True*, *timeout=None*)
Remove and return an item from the queue.

If optional args 'block' is true and 'timeout' is None (the default), block if necessary until an item is available. If 'timeout' is a non-negative number, it blocks at most 'timeout' seconds and raises the Empty exception if no item was available within that time. Otherwise ('block' is false), return an item if one is immediately available, else raise the Empty exception ('timeout' is ignored in that case).

get_nowait()
Remove and return an item from the queue without blocking.

Only get an item if one is immediately available. Otherwise raise the Empty exception.

join()
Blocks until all items in the Queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls task_done() to indicate the item was retrieved and all work on it is complete.

When the count of unfinished tasks drops to zero, join() unblocks.

put(*item*, *block=True*, *timeout=None*)
Put an item into the queue.

If optional args 'block' is true and 'timeout' is None (the default), block if necessary until a free slot is available. If 'timeout' is a non-negative number, it blocks at most 'timeout' seconds and raises the Full exception if no free slot was available within that time. Otherwise ('block' is false), put an item on the queue if a free slot is immediately available, else raise the Full exception ('timeout' is ignored in that case).

put_nowait(*item*)
Put an item into the queue without blocking.

Only enqueue the item if a free slot is immediately available. Otherwise raise the Full exception.

qsize()
Return the approximate size of the queue (not reliable!).

task_done()
Indicate that a formerly enqueued task is complete.

Used by Queue consumer threads. For each get() used to fetch a task, a subsequent call to task_done() tells the queue that the processing on the task is complete.

If a join() is currently blocking, it will resume when all items have been processed (meaning that a task_done() call was received for every item that had been put() into the queue).

Raises a ValueError if called more times than there were items placed in the queue.

class Semaphore(*value=1*)
Bases: `object`

This class implements semaphore objects.

Semaphores manage a counter representing the number of release() calls minus the number of acquire() calls, plus an initial value. The acquire() method blocks if necessary until it can return without making the counter negative. If not given, value defaults to 1.

Copyright (c) 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015 Python Software Foundation. All rights reserved.

__enter__(*blocking=True*, *timeout=None*)
Acquire a semaphore, decrementing the internal counter by one.

When invoked without arguments: if the internal counter is larger than zero on entry, decrement it by one and return immediately. If it is zero on entry, block, waiting until some other thread has called release() to make it larger than zero. This is done with proper interlocking so that if multiple acquire() calls are blocked, release() will wake exactly one of them up. The implementation may pick one at random, so the order in which blocked threads are awakened should not be relied on. There is no return value in this case.

When invoked with blocking set to true, do the same thing as when called without arguments, and return true.

When invoked with blocking set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

When invoked with a timeout other than None, it will block for at most timeout seconds. If acquire does not complete successfully in that interval, return false. Return true otherwise.

__init__(*value=1*)
x.__init__(...) initializes x; see help(type(x)) for signature

__weakref__
list of weak references to the object (if defined)

acquire(*blocking=True*, *timeout=None*)
Acquire a semaphore, decrementing the internal counter by one.

When invoked without arguments: if the internal counter is larger than zero on entry, decrement it by one and return immediately. If it is zero on entry, block, waiting until some other thread has called release() to make it larger than zero. This is done with proper interlocking so that if multiple acquire() calls are blocked, release() will wake exactly one of them up. The implementation may pick one at random, so the order in which blocked threads are awakened should not be relied on. There is no return value in this case.

When invoked with blocking set to true, do the same thing as when called without arguments, and return true.

When invoked with blocking set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

When invoked with a timeout other than None, it will block for at most timeout seconds. If acquire does not complete successfully in that interval, return false. Return true otherwise.

release()
Release a semaphore, incrementing the internal counter by one.

When the counter is zero on entry and another thread is waiting for it to become larger than zero again, wake up that thread.

SockErr
alias of `socket.error`

Socket = <module 'socket' from '/usr/lib/python2.7/socket.pyc'>

spawn(*target*, **args*, ***kwargs*)
Create the worker that will process the work to be handled

class pykafka.handlers.**RequestHandler**(*handler*, *connection*)
Bases: `object`

Uses a Handler instance to dispatch requests.

class Shared(*connection*, *requests*, *ending*)
Bases: `tuple`

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *connection*, *requests*, *ending*)
Create new instance of Shared(connection, requests, ending)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0x906d60>*, *len=<built-in function len>*)
Make a new Shared object from a sequence or iterable

_replace(***kwds*)
Return a new Shared object replacing specified fields with new values

connection
Alias for field number 0

ending
Alias for field number 2

requests
Alias for field number 1

class Task(*request*, *future*)
Bases: `tuple`

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *request*, *future*)
Create new instance of Task(request, future)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0x906d60>*, *len=<built-in function len>*)
Make a new Task object from a sequence or iterable

_replace(***kwds*)
Return a new Task object replacing specified fields with new values

future
Alias for field number 1

request
Alias for field number 0

__init__(*handler*, *connection*)

__weakref__
list of weak references to the object (if defined)

_start_thread()
Run the request processor

request(*request*, *has_response=True*)
Construct a new request

Parameters **has_response** – Whether this request will return a response

Returns *pykafka.handlers.ResponseFuture*

start()
Start the request processor.

stop()
Stop the request processor.

6.2.9 pykafka.managedbalancedconsumer

class pykafka.managedbalancedconsumer.**ManagedBalancedConsumer**(*topic*, *cluster*, *consumer_group*, *fetch_message_max_bytes=1048576*, *num_consumer_fetchers=1*, *auto_commit_enable=False*, *auto_commit_interval_ms=60000*, *queued_max_messages=2000*, *fetch_min_bytes=1*, *fetch_error_backoff_ms=500*, *fetch_wait_max_ms=100*, *offsets_channel_backoff_ms=1000*, *offsets_commit_max_retries=5*, *auto_offset_reset=-2*, *consumer_timeout_ms=-1*, *rebalance_max_retries=5*, *rebalance_backoff_ms=2000*, *auto_start=True*, *reset_offset_on_start=False*, *post_rebalance_callback=None*, *use_rdkafka=False*, *compacted_topic=True*, *heartbeat_interval_ms=3000*, *membership_protocol=GroupMembershipProtocol*, *protocol_name='range'*, *metadata=<pykafka.protocol.group_membersh object>*, *decide_partitions=<function decide_partitions_range>)*, *deserializer=None*, *reset_offset_on_fetch=True*)

Bases: *pykafka.balancedconsumer.BalancedConsumer*

A self-balancing consumer that uses Kafka 0.9's Group Membership API

Implements the Group Management API semantics for Kafka 0.9 compatibility

Maintains a single instance of SimpleConsumer, periodically using the consumer rebalancing algorithm to reassign partitions to this SimpleConsumer.

This class overrides the functionality of *pykafka.balancedconsumer.BalancedConsumer* that deals with ZooKeeper and inherits other functionality directly.

__init__(*topic*, *cluster*, *consumer_group*, *fetch_message_max_bytes=1048576*, *num_consumer_fetchers=1*, *auto_commit_enable=False*, *auto_commit_interval_ms=60000*, *queued_max_messages=2000*, *fetch_min_bytes=1*, *fetch_error_backoff_ms=500*, *fetch_wait_max_ms=100*, *offsets_channel_backoff_ms=1000*, *offsets_commit_max_retries=5*, *auto_offset_reset=-2*, *consumer_timeout_ms=-1*, *rebalance_max_retries=5*, *rebalance_backoff_ms=2000*, *auto_start=True*, *reset_offset_on_start=False*, *post_rebalance_callback=None*, *use_rdkafka=False*, *compacted_topic=True*, *heartbeat_interval_ms=3000*, *membership_protocol=GroupMembershipProtocol(protocol_type='consumer'*, *protocol_name='range'*, *metadata=<pykafka.protocol.group_membership.ConsumerGroupProtocolMetadata object>*, *decide_partitions=<function decide_partitions_range>)*, *deserializer=None*, *reset_offset_on_fetch=True*)

Create a ManagedBalancedConsumer instance

Parameters

- **topic** (*pykafka.topic.Topic*) – The topic this consumer should consume
- **cluster** (*pykafka.cluster.Cluster*) – The cluster to which this consumer should connect
- **consumer_group** (*str*) – The name of the consumer group this consumer should join. Consumer group names are namespaced at the cluster level, meaning that two consumers consuming different topics with the same group name will be treated as part of the same group.
- **fetch_message_max_bytes** (*int*) – The number of bytes of messages to attempt to fetch with each fetch request
- **num_consumer_fetchers** (*int*) – The number of workers used to make FetchRequests
- **auto_commit_enable** (*bool*) – If true, periodically commit to kafka the offset of messages already fetched by this consumer. This also requires that *consumer_group* is not *None*.
- **auto_commit_interval_ms** (*int*) – The frequency (in milliseconds) at which the consumer's offsets are committed to kafka. This setting is ignored if *auto_commit_enable* is *False*.
- **queued_max_messages** (*int*) – The maximum number of messages buffered for consumption in the internal *pykafka.simpleconsumer.SimpleConsumer*
- **fetch_min_bytes** (*int*) – The minimum amount of data (in bytes) that the server should return for a fetch request. If insufficient data is available, the request will block until sufficient data is available.
- **fetch_error_backoff_ms** (*int*) – *UNUSED*. See *pykafka.simpleconsumer.SimpleConsumer*.

- **fetch_wait_max_ms** (*int*) – The maximum amount of time (in milliseconds) that the server will block before answering a fetch request if there isn't sufficient data to immediately satisfy *fetch_min_bytes*.
- **offsets_channel_backoff_ms** (*int*) – Backoff time to retry failed offset commits and fetches.
- **offsets_commit_max_retries** (*int*) – The number of times the offset commit worker should retry before raising an error.
- **auto_offset_reset** (*pykafka.common.OffsetType*) – What to do if an offset is out of range. This setting indicates how to reset the consumer's internal offset counter when an *OffsetOutOfRangeError* is encountered.
- **consumer_timeout_ms** (*int*) – Amount of time (in milliseconds) the consumer may spend without messages available for consumption before returning None.
- **rebalance_max_retries** (*int*) – The number of times the rebalance should retry before raising an error.
- **rebalance_backoff_ms** (*int*) – Backoff time (in milliseconds) between retries during rebalance.
- **auto_start** (*bool*) – Whether the consumer should start after __init__ is complete. If false, it can be started with *start()*.
- **reset_offset_on_start** (*bool*) – Whether the consumer should reset its internal offset counter to *self._auto_offset_reset* and commit that offset immediately upon starting up
- **post_rebalance_callback** (*function*) – A function to be called when a rebalance is in progress. This function should accept three arguments: the *pykafka.balancedconsumer.BalancedConsumer* instance that just completed its rebalance, a dict of partitions that it owned before the rebalance, and a dict of partitions it owns after the rebalance. These dicts map partition ids to the most recently known offsets for those partitions. This function can optionally return a dictionary mapping partition ids to offsets. If it does, the consumer will reset its offsets to the supplied values before continuing consumption. Note that the BalancedConsumer is in a poorly defined state at the time this callback runs, so that accessing its properties (such as *held_offsets* or *partitions*) might yield confusing results. Instead, the callback should really rely on the provided partition-id dicts, which are well-defined.
- **use_rdkafka** (*bool*) – Use librdkafka-backed consumer if available
- **compacted_topic** (*bool*) – Set to read from a compacted topic. Forces consumer to use less stringent message ordering logic because compacted topics do not provide offsets in strict incrementing order.
- **heartbeat_interval_ms** (*int*) – The amount of time in milliseconds to wait between heartbeat requests
- **membership_protocol** (*pykafka.membershipprotocol.GroupMembershipProtocol*) – The group membership protocol to which this consumer should adhere
- **deserializer** (*function*) – A function defining how to deserialize messages returned from Kafka. A function with the signature d(value, partition_key) that returns a tuple of (deserialized_value, deserialized_partition_key). The arguments passed to this function are the bytes representations of a message's value and partition key, and the returned data should be these fields transformed according to the client code's serialization logic. See *pykafka.utils.__init__* for stock implemtations.

- **reset_offset_on_fetch** (*bool*) – Whether to update offsets during fetch_offsets. Disable for read-only use cases to prevent side-effects.

_build_default_error_handlers()
Set up default responses to common error codes

_handle_error(*error_code*)
Call the appropriate handler function for the given error code

Parameters **error_code** (*int*) – The error code returned from a Group Membership API request

_join_group()
Send a JoinGroupRequest.

Assigns a member id and tells the coordinator about this consumer.

_setup_heartbeat_worker()
Start the heartbeat worker

_sync_group(*group_assignments*)
Send a SyncGroupRequest.

If this consumer is the group leader, this call informs the other consumers of their partition assignments. For all consumers including the leader, this call is used to fetch partition assignments.

The group leader *could* tell itself its own assignment instead of using the result of this request, but it does the latter to ensure consistency.

_update_member_assignment()
Join a managed consumer group and start consuming assigned partitions

Equivalent to *pykafka.balancedconsumer.BalancedConsumer._update_member_assignment*, but uses the Kafka 0.9 Group Membership API instead of ZooKeeper to manage group state

start()
Start this consumer.

Must be called before consume() if *auto_start=False*.

stop()
Stop this consumer

Should be called as part of a graceful shutdown

6.2.10 pykafka.membershipprotocol

class pykafka.membershipprotocol.**GroupMembershipProtocol**(*protocol_type*, *protocol_name*, *metadata*, *decide_partitions*)

Bases: tuple

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *protocol_type*, *protocol_name*, *metadata*, *decide_partitions*)
Create new instance of GroupMembershipProtocol(protocol_type, protocol_name, metadata, decide_partitions)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0x906d60>*, *len=<built-in function len>*)
Make a new GroupMembershipProtocol object from a sequence or iterable

_replace(***kwds*)
Return a new GroupMembershipProtocol object replacing specified fields with new values

decide_partitions
Alias for field number 3

metadata
Alias for field number 2

protocol_name
Alias for field number 1

protocol_type
Alias for field number 0

pykafka.membershipprotocol.**decide_partitions_range**(*participants*, *partitions*, *consumer_id*)
Decide which partitions belong to this consumer_id.

Uses the consumer rebalancing algorithm described here https://kafka.apache.org/documentation/#impl_consumerrebalance

It is very important that the participants array is sorted, since this algorithm runs on each consumer and indexes into the same array. The same array index operation must return the same result on each consumer.

Parameters

- **participants** (Iterable of *bytes*) – Sorted list of ids of all consumers in this consumer group.
- **partitions** (Iterable of *pykafka.partition.Partition*) – List of all partitions on the topic being consumed
- **consumer_id** (*bytes*) – The ID of the consumer for which to generate a partition assignment.

pykafka.membershipprotocol.**decide_partitions_roundrobin**(*participants*, *partitions*, *consumer_id*)
Decide which partitions belong to this consumer_id.

Uses the "roundrobin" strategy described here https://kafka.apache.org/documentation/#oldconsumerconfigs

Parameters

- **participants** (Iterable of *bytes*) – Sorted list of ids of all consumers in this consumer group.
- **partitions** (Iterable of *pykafka.partition.Partition*) – List of all partitions on the topic being consumed
- **consumer_id** (*bytes*) – The ID of the consumer for which to generate a partition assignment.

6.2.11 pykafka.partition

Author: Keith Bourgoin, Emmett Butler

class pykafka.partition.**Partition**(*topic*, *id_*, *leader*, *replicas*, *isr*)
Bases: object

A Partition is an abstraction over the kafka concept of a partition. A kafka partition is a logical division of the logs for a topic. Its messages are totally ordered.

__eq__(*other*)
x.__eq__(y) <==> x==y

__hash__() <==> *hash(x)*

__init__(*topic*, *id_*, *leader*, *replicas*, *isr*)
Instantiate a new Partition

Parameters

- **topic** (*pykafka.topic.Topic*) – The topic to which this Partition belongs
- **id** (*int*) – The identifier for this partition
- **leader** (*pykafka.broker.Broker*) – The broker that is currently acting as the leader for this partition.
- **replicas** (Iterable of *pykafka.broker.Broker*) – A list of brokers containing this partition's replicas
- **isr** (*pykafka.broker.Broker*) – The current set of in-sync replicas for this partition

__lt__(*other*)
x.__lt__(y) <==> x<y

__ne__(*other*)
x.__ne__(y) <==> x!=y

__repr__() <==> *repr(x)*

__weakref__
list of weak references to the object (if defined)

earliest_available_offset()
Get the earliest offset for this partition.

fetch_offset_limit(*offsets_before*, *max_offsets=1*)

Use the Offset API to find a limit of valid offsets for this partition.

Parameters

- **offsets_before** (*datetime.datetime* or int) – Return an offset from before this timestamp (in milliseconds). Deprecated::2.7,3.6: do not use int
- **max_offsets** (*int*) – The maximum number of offsets to return

id
The identifying int for this partition, unique within its topic

isr
The current list of in-sync replicas for this partition

latest_available_offset()
Get the offset of the next message that would be appended to this partition

leader
The broker currently acting as leader for this partition

replicas
The list of brokers currently holding replicas of this partition

topic
The topic to which this partition belongs

update(*brokers*, *metadata*)
Update this partition with fresh metadata.

Parameters

- **brokers** (List of *pykafka.broker.Broker*) – Brokers on which partitions exist
- **metadata** (pykafka.protocol.PartitionMetadata) – Metadata for the partition

6.2.12 pykafka.partitioners

Author: Keith Bourgoin, Emmett Butler

class pykafka.partitioners.**RandomPartitioner**
Bases: *pykafka.partitioners.BasePartitioner*

Returns a random partition out of all of the available partitions.

Uses a non-random incrementing counter to provide even distribution across partitions without wasting CPU cycles

__call__(*...*) <==> *x(...)*

__init__()
x.__init__(...) initializes x; see help(type(x)) for signature

class pykafka.partitioners.**BasePartitioner**
Bases: object

Base class for custom class-based partitioners.

A partitioner is used by the *pykafka.producer.Producer* to decide which partition to which to produce messages.

__call__(*...*) <==> *x(...)*

__weakref__
list of weak references to the object (if defined)

class pykafka.partitioners.**HashingPartitioner**(*hash_func=None*)
Bases: *pykafka.partitioners.BasePartitioner*

Returns a (relatively) consistent partition out of all available partitions based on the key.

Messages that are published with the same keys are not guaranteed to end up on the same broker if the number of brokers changes (due to the addition or removal of a broker, planned or unplanned) or if the number of topics per partition changes. This is also unreliable when not all brokers are aware of a topic, since the number of available partitions will be in flux until all brokers have accepted a write to that topic and have declared how many partitions that they are actually serving.

__call__(*partitions*, *key*)

Parameters

- **partitions** (sequence of `pykafka.base.BasePartition`) – The partitions from which to choose
- **key** (Any hashable type if using the default `hash()` implementation, any valid value for your custom hash function) – Key used for routing

Returns A partition

Return type `pykafka.base.BasePartition`

__init__(*hash_func=None*)

Parameters **hash_func** (*function*) – hash function (defaults to `hash()`), should return an *int*. If hash randomization (Python 2.7) is enabled, a custom hashing function should be defined that is consistent between interpreter restarts.

class pykafka.partitioners.**GroupHashingPartitioner**(*hash_func*, *group_size=1*)

Bases: *pykafka.partitioners.BasePartitioner*

Messages published with the identical keys will be directed to a consistent subset of 'n' partitions from the set of available partitions. For example, if there are 16 partitions and group_size=4, messages with the identical keys will be shared equally between a subset of four partitions, instead of always being directed to the same partition.

The same guarantee caveats apply as to the `pykafka.base.HashingPartitioner`.

__call__(*partitions*, *key*)

Parameters

- **partitions** (sequence of `pykafka.base.BasePartition`) – The partitions from which to choose
- **key** (Any hashable type if using the default `hash()` implementation, any valid value for your custom hash function) – Key used for routing

Returns A partition

Return type `pykafka.base.BasePartition`

__init__(*hash_func*, *group_size=1*)

Parameters

- **hash_func** (*function*) – A hash function
- **group_size** (*Integer value between (0, total_partition_count)*) – Size of the partition group to assign to. For example, if there are 16 partitions, and we want to smooth the distribution of identical keys between a set of 4, use 4 as the group_size.

6.2.13 pykafka.producer

class pykafka.producer.**Producer**(*cluster*, *topic*, *partitioner=None*, *compression=0*, *max_retries=3*, *retry_backoff_ms=100*, *required_acks=1*, *ack_timeout_ms=10000*, *max_queued_messages=100000*, *min_queued_messages=70000*, *linger_ms=5000*, *queue_empty_timeout_ms=0*, *block_on_queue_full=True*, *max_request_size=1000012*, *sync=False*, *delivery_reports=False*, *pending_timeout_ms=5000*, *auto_start=True*, *serializer=None*)

Bases: `object`

Implements asynchronous producer logic similar to the JVM driver.

It creates a thread of execution for each broker that is the leader of one or more of its topic's partitions. Each of these threads (which may use *threading* or some other parallelism implementation like *gevent*) is associated with a queue that holds the messages that are waiting to be sent to that queue's broker.

__enter__()
Context manager entry point - start the producer

__exit__(*exc_type*, *exc_value*, *traceback*)
Context manager exit point - stop the producer

__init__(*cluster*, *topic*, *partitioner=None*, *compression=0*, *max_retries=3*, *retry_backoff_ms=100*, *required_acks=1*, *ack_timeout_ms=10000*, *max_queued_messages=100000*, *min_queued_messages=70000*, *linger_ms=5000*, *queue_empty_timeout_ms=0*, *block_on_queue_full=True*, *max_request_size=1000012*, *sync=False*, *delivery_reports=False*, *pending_timeout_ms=5000*, *auto_start=True*, *serializer=None*)
Instantiate a new AsyncProducer

Parameters

- **cluster** (*pykafka.cluster.Cluster*) – The cluster to which to connect
- **topic** (*pykafka.topic.Topic*) – The topic to which to produce messages
- **partitioner** (*pykafka.partitioners.BasePartitioner*) – The partitioner to use during message production
- **compression** (*pykafka.common.CompressionType*) – The type of compression to use.
- **max_retries** (*int*) – How many times to attempt to produce a given batch of messages before raising an error. Allowing retries will potentially change the ordering of records because if two records are sent to a single partition, and the first fails and is retried but the second succeeds, then the second record may appear first. If you want to completely disallow message reordering, use *sync=True*.
- **retry_backoff_ms** (*int*) – The amount of time (in milliseconds) to back off during produce request retries. This does not equal the total time spent between message send attempts, since that number can be influenced by other kwargs, including *linger_ms* and *socket_timeout_ms*.
- **required_acks** (*int*) – The number of other brokers that must have committed the data to their log and acknowledged this to the leader before a request is considered complete
- **ack_timeout_ms** (*int*) – The amount of time (in milliseconds) to wait for acknowledgment of a produce request on the server.
- **max_queued_messages** (*int*) – The maximum number of messages the producer can have waiting to be sent to the broker. If messages are sent faster than they can be delivered to the broker, the producer will either block or throw an exception based on the preference specified with block_on_queue_full.
- **min_queued_messages** (*int*) – The minimum number of messages the producer can have waiting in a queue before it flushes that queue to its broker (must be greater than 0). This paramater can be used to control the number of messages sent in one batch during async production. This parameter is automatically overridden to 1 when *sync=True*.
- **linger_ms** (*int*) – This setting gives the upper bound on the delay for batching: once the producer gets min_queued_messages worth of messages for a broker, it will be sent immediately regardless of this setting. However, if we have fewer than this many messages

accumulated for this partition we will 'linger' for the specified time waiting for more records to show up. linger_ms=0 indicates no lingering - messages are sent as fast as possible after they are 'produce()'d.

- **queue_empty_timeout_ms** (*int*) – The amount of time in milliseconds for which the producer's worker threads should block when no messages are available to flush to brokers. After each *linger_ms* interval, the worker thread checks for the presence of at least one message in its queue. If there is not at least one, it enters an "empty wait" period for *queue_empty_timeout_ms* before starting a new *linger_ms* wait loop. If *queue_empty_timeout_ms* is 0, this "empty wait" period is a noop, and flushes will continue to be attempted at intervals of *linger_ms*, even when the queue is empty. If *queue_empty_timeout_ms* is a positive integer, this "empty wait" period will last for at most that long, but it ends earlier if a message is *produce()'d before that time. If 'queue_empty_timeout_ms* is -1, the "empty wait" period can only be stopped (and the worker thread killed) by a call to either *produce()* or *stop()* - it will never time out.
- **block_on_queue_full** (*bool*) – When the producer's message queue for a broker contains max_queued_messages, we must either stop accepting new messages (block) or throw an error. If True, this setting indicates we should block until space is available in the queue. If False, we should throw an error immediately.
- **max_request_size** (*int*) – The maximum size of a request in bytes. This is also effectively a cap on the maximum record size. Note that the server has its own cap on record size which may be different from this. This setting will limit the number of record batches the producer will send in a single request to avoid sending huge requests.
- **sync** (*bool*) – Whether calls to *produce* should wait for the message to send before returning. If *True*, an exception will be raised from *produce()* if delivery to kafka failed.
- **delivery_reports** (*bool*) – If set to *True*, the producer will maintain a thread-local queue on which delivery reports are posted for each message produced. These must regularly be retrieved through *get_delivery_report()*, which returns a 2-tuple of *pykafka.protocol.Message* and either *None* (for success) or an *Exception* in case of failed delivery to kafka. If *get_delivery_report()* is not called regularly with this setting enabled, memory usage will grow unbounded. This setting is ignored when *sync=True*.
- **pending_timeout_ms** – The amount of time (in milliseconds) to wait for delivery reports to be returned from the broker during a *produce()* call. Also, the time in ms to wait during a *stop()* call for all messages to be marked as delivered. -1 indicates that these calls should block indefinitely. Differs from *ack_timeout_ms* in that *ack_timeout_ms* is a value sent to the broker to control the broker-side timeout, while *pending_timeout_ms* is used internally by pykafka and not sent to the broker.
- **auto_start** (*bool*) – Whether the producer should begin communicating with kafka after __init__ is complete. If false, communication can be started with *start()*.
- **serializer** (*function*) – A function defining how to serialize messages to be sent to Kafka. A function with the signature d(value, partition_key) that returns a tuple of (serialized_value, serialized_partition_key). The arguments passed to this function are a message's value and partition key, and the returned data should be these fields transformed according to the client code's serialization logic. See *pykafka.utils.__init__* for stock implemtations.

__repr__(*)* <==> *repr(x)*

__weakref__
list of weak references to the object (if defined)

_produce(*message*)

Enqueue a message for the relevant broker Attempts to update metadata in response to missing brokers. :param message: Message with valid *partition_id*, ready to be sent :type message: *pykafka.protocol.Message*

_produce_has_timed_out(*start_time*)
Indicates whether enough time has passed since start_time for a *produce()* call to timeout

_raise_worker_exceptions()
Raises exceptions encountered on worker threads

_send_request(*message_batch*, *owned_broker*)
Send the produce request to the broker and handle the response.

Parameters

- **message_batch** (iterable of *pykafka.protocol.Message*) – An iterable of messages to send
- **owned_broker** (`pykafka.producer.OwnedBroker`) – The broker to which to send the request

_setup_owned_brokers()
Instantiate one OwnedBroker per broker

If there are already OwnedBrokers instantiated, safely stop and flush them before creating new ones.

_update()
Update the producer and cluster after an ERROR_CODE

Also re-produces messages that were in queues at the time the update was triggered

_wait_all()
Block until all pending messages are sent or until pending_timeout_ms

"Pending" messages are those that have been used in calls to *produce* and have not yet been acknowledged in a response from the broker

get_delivery_report(*block=True*, *timeout=None*)
Fetch delivery reports for messages produced on the current thread

Returns 2-tuples of a *pykafka.protocol.Message* and either *None* (for successful deliveries) or *Exception* (for failed deliveries). This interface is only available if you enabled *delivery_reports* on init (and you did not use *sync=True*)

Parameters

- **block** (*bool*) – Whether to block on dequeueing a delivery report
- **timeout** – How long (in seconds) to block before returning None

;type timeout: int

produce(*message*, *partition_key=None*, *timestamp=None*)
Produce a message.

Parameters

- **message** (*bytes*) – The message to produce (use None to send null)
- **partition_key** (*bytes*) – The key to use when deciding which partition to send this message to. This key is passed to the *partitioner*, which may or may not use it in deciding the partition. The default *RandomPartitioner* does not use this key, but the optional *HashingPartitioner* does.

- **timestamp** (*datetime.datetime*) – The timestamp at which the message is produced (requires broker_version >= 0.10.0)

Returns The *pykafka.protocol.Message* instance that was added to the internal message queue

start()
Set up data structures and start worker threads

stop()
Mark the producer as stopped, and wait until all messages to be sent

6.2.14 pykafka.protocol

class pykafka.protocol.**MetadataRequest**(*topics=None*, **kwargs*)
Bases: pykafka.protocol.base.Request

Metadata Request Specification:

```
MetadataRequest => [TopicName]
    TopicName => string
```

__init__(*topics=None*, **kwargs*)
Create a new MetadataRequest :param topics: Topics to query. Leave empty for all available topics.

__len__()
Length of the serialized message, in bytes

get_bytes()
Serialize the message :returns: Serialized message :rtype: bytearray

class pykafka.protocol.**MetadataResponse**(*buff*)
Bases: pykafka.protocol.base.Response

Response from MetadataRequest Specification:: Metadata Response (Version: 0) => [brokers] [topic_metadata]

brokers => node_id host port node_id => INT32 host => STRING port => INT32

topic_metadata => error_code topic [partition_metadata] error_code => INT16 topic => STRING partition_metadata => error_code partition leader [replicas] [isr]

error_code => INT16 partition => INT32 leader => INT32 replicas => INT32 isr => INT32

__init__(*buff*)
Deserialize into a new Response :param buff: Serialized message :type buff: bytearray

class pykafka.protocol.**ProduceRequest**(*compression_type=0*, *required_acks=1*, *timeout=10000*, *broker_version='0.9.0'*)
Bases: pykafka.protocol.base.Request

Produce Request Specification:

```
ProduceRequest => RequiredAcks Timeout [TopicName [Partition MessageSetSize␣
↪MessageSet]]
  RequiredAcks => int16
  Timeout => int32
  Partition => int32
  MessageSetSize => int32
```

__init__(*compression_type=0*, *required_acks=1*, *timeout=10000*, *broker_version='0.9.0'*)
Create a new ProduceRequest `required_acks` determines how many acknowledgement the server waits for before returning. This is useful for ensuring the replication factor of published messages. The behavior is:

```
-1: Block until all servers acknowledge
0: No waiting -- server doesn't even respond to the Produce request
1: Wait for this server to write to the local log and then return
2+: Wait for N servers to acknowledge
```

Parameters

- **partition_requests** – Iterable of `kafka.pykafka.protocol.PartitionProduceRequest` for this request
- **compression_type** – Compression to use for messages
- **required_acks** – see docstring
- **timeout** – timeout (in ms) to wait for the required acks

__len__()
Length of the serialized message, in bytes

add_message(*message*, *topic_name*, *partition_id*)
Add a list of `kafka.common.Message` to the waiting request :param messages: an iterable of `kafka.common.Message` to add :param topic_name: the name of the topic to publish to :param partition_id: the partition to publish to

get_bytes()
Serialize the message :returns: Serialized message :rtype: `bytearray`

message_count()
Get the number of messages across all MessageSets in the request.

messages
Iterable of all messages in the Request

class pykafka.protocol.**ProduceResponse**(*buff*)
Bases: `pykafka.protocol.base.Response`

Produce Response. Checks to make sure everything went okay. Specification:

```
ProduceResponse => [TopicName [Partition ErrorCode Offset]]
  TopicName => string
  Partition => int32
  ErrorCode => int16
  Offset => int64
```

__init__(*buff*)
Deserialize into a new Response :param buff: Serialized message :type buff: `bytearray`

class pykafka.protocol.**PartitionFetchRequest**
Bases: `pykafka.protocol.fetch.PartitionFetchRequest`

Fetch request for a specific topic/partition

Variables

- **topic_name** – Name of the topic to fetch from
- **partition_id** – Id of the partition to fetch from

- ***offset*** – Offset at which to start reading
- **max_bytes** – Max bytes to read from this partition (default: 300kb)

class pykafka.protocol.**FetchRequest**(*partition_requests=[]*, *timeout=1000*, *min_bytes=1024*, *api_version=0*)

Bases: pykafka.protocol.base.Request

A Fetch request sent to Kafka

Specification:

```
FetchRequest => ReplicaId MaxWaitTime MinBytes [TopicName [Partition FetchOffset␣
↪MaxBytes]]
  ReplicaId => int32
  MaxWaitTime => int32
  MinBytes => int32
  TopicName => string
  Partition => int32
  FetchOffset => int64
  MaxBytes => int32
```

__init__(*partition_requests=[]*, *timeout=1000*, *min_bytes=1024*, *api_version=0*)

Create a new fetch request

Kafka 0.8 uses long polling for fetch requests, which is different from 0.7x. Instead of polling and waiting, we can now set a timeout to wait and a minimum number of bytes to be collected before it returns. This way we can block effectively and also ensure good network throughput by having fewer, large transfers instead of many small ones every time a byte is written to the log.

Parameters

- **partition_requests** – Iterable of kafka.pykafka..protocol.PartitionFetchRequest for this request
- **timeout** – Max time to wait (in ms) for a response from the server
- **min_bytes** – Minimum bytes to collect before returning

__len__()

Length of the serialized message, in bytes

add_request(*partition_request*)

Add a topic/partition/offset to the requests

Parameters

- **topic_name** – The topic to fetch from
- **partition_id** – The partition to fetch from
- **offset** – The offset to start reading data from
- **max_bytes** – The maximum number of bytes to return in the response

get_bytes()

Serialize the message

Returns Serialized message

Return type bytearray

class pykafka.protocol.**FetchPartitionResponse**(*max_offset*, *messages*, *err*)

Bases: tuple

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *max_offset*, *messages*, *err*)
Create new instance of FetchPartitionResponse(max_offset, messages, err)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0x906d60>*, *len=<built-in function len>*)
Make a new FetchPartitionResponse object from a sequence or iterable

_replace(***kwds*)
Return a new FetchPartitionResponse object replacing specified fields with new values

err
Alias for field number 2

max_offset
Alias for field number 0

messages
Alias for field number 1

class pykafka.protocol.**FetchResponse**(*buff*, *offset=0*, *broker_version='0.9.0'*)
Bases: pykafka.protocol.base.Response

Unpack a fetch response from the server

Specification:

```
FetchResponse => [TopicName [Partition ErrorCode HighwaterMarkOffset␣
↪MessageSetSize MessageSet]]
  TopicName => string
  Partition => int32
  ErrorCode => int16
  HighwaterMarkOffset => int64
  MessageSetSize => int32
```

__init__(*buff*, *offset=0*, *broker_version='0.9.0'*)
Deserialize into a new Response

Parameters

- **buff** (bytearray) – Serialized message
- **offset** (*int*) – Offset into the message

_unpack_message_set(*buff*, *partition_id=-1*, *broker_version='0.9.0'*)
MessageSets can be nested. Get just the Messages out of it.

class pykafka.protocol.**ListOffsetRequest**(*partition_requests*)
Bases: pykafka.protocol.base.Request

An offset request Specification:

```
ListOffsetRequest => ReplicaId [TopicName [Partition Time MaxNumberOfOffsets]]
  ReplicaId => int32
  TopicName => string
  Partition => int32
  Time => int64
  MaxNumberOfOffsets => int32
```

__init__(*partition_requests*)
Create a new offset request

__len__()
Length of the serialized message, in bytes

get_bytes()
Serialize the message :returns: Serialized message :rtype: bytearray

class pykafka.protocol.**ListOffsetResponse**(*buff*)
Bases: pykafka.protocol.base.Response

An offset response Specification:

```
ListOffsetResponse => [TopicName [PartitionOffsets]]
  PartitionOffsets => Partition ErrorCode [Offset]
  Partition => int32
  ErrorCode => int16
  Offset => int64
```

__init__(*buff*)
Deserialize into a new Response :param buff: Serialized message :type buff: bytearray

class pykafka.protocol.**GroupCoordinatorRequest**(*consumer_group*)
Bases: pykafka.protocol.base.Request

A consumer metadata request Specification:

```
GroupCoordinatorRequest => ConsumerGroup
    ConsumerGroup => string
```

__init__(*consumer_group*)
Create a new group coordinator request

__len__()
Length of the serialized message, in bytes

get_bytes()
Serialize the message :returns: Serialized message :rtype: bytearray

class pykafka.protocol.**GroupCoordinatorResponse**(*buff*)
Bases: pykafka.protocol.base.Response

A group coordinator response Specification:

```
GroupCoordinatorResponse => ErrorCode CoordinatorId CoordinatorHost␣
↪CoordinatorPort
    ErrorCode => int16
    CoordinatorId => int32
    CoordinatorHost => string
    CoordinatorPort => int32
```

__init__(*buff*)
Deserialize into a new Response :param buff: Serialized message :type buff: bytearray

class pykafka.protocol.**PartitionOffsetCommitRequest**
Bases: pykafka.protocol.offset_commit.PartitionOffsetCommitRequest

Offset commit request for a specific topic/partition :ivar topic_name: Name of the topic to look up :ivar partition_id: Id of the partition to look up :ivar offset: :ivar timestamp: :ivar metadata: arbitrary metadata that should be committed with this offset commit

class pykafka.protocol.**OffsetCommitRequest**(*consumer_group*, *consumer_group_generation_id*, *consumer_id*, *partition_requests=[]*)
Bases: pykafka.protocol.base.Request

An offset commit request Specification:

```
OffsetCommitRequest => ConsumerGroupId ConsumerGroupGenerationId ConsumerId␣
↪[TopicName [Partition Offset TimeStamp Metadata]]
    ConsumerGroupId => string
    ConsumerGroupGenerationId => int32
    ConsumerId => string
    TopicName => string
    Partition => int32
    Offset => int64
    TimeStamp => int64
    Metadata => string
```

__init__(*consumer_group*, *consumer_group_generation_id*, *consumer_id*, *partition_requests=[]*)
Create a new offset commit request :param partition_requests: Iterable of

kafka.pykafka.protocol.PartitionOffsetCommitRequest for this request

__len__()
Length of the serialized message, in bytes

get_bytes()
Serialize the message :returns: Serialized message :rtype: bytearray

class pykafka.protocol.**OffsetCommitPartitionResponse**(*err*)
Bases: tuple

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *err*)
Create new instance of OffsetCommitPartitionResponse(err,)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0x906d60>*, *len=<built-in function len>*)
Make a new OffsetCommitPartitionResponse object from a sequence or iterable

_replace(***kwds*)
Return a new OffsetCommitPartitionResponse object replacing specified fields with new values

err
Alias for field number 0

class pykafka.protocol.**OffsetCommitResponse**(*buff*)
Bases: pykafka.protocol.base.Response

An offset commit response Specification:

```
OffsetCommitResponse => [TopicName [Partition ErrorCode]]]
    TopicName => string
    Partition => int32
    ErrorCode => int16
```

__init__(*buff*)
Deserialize into a new Response :param buff: Serialized message :type buff: bytearray

class pykafka.protocol.**PartitionOffsetFetchRequest**
Bases: pykafka.protocol.offset_commit.PartitionOffsetFetchRequest

Offset fetch request for a specific topic/partition :ivar topic_name: Name of the topic to look up :ivar partition_id: Id of the partition to look up

class pykafka.protocol.**OffsetFetchRequest**(*consumer_group*, *partition_requests=[]*)
Bases: pykafka.protocol.base.Request

An offset fetch request Specification:

```
OffsetFetchRequest => ConsumerGroup [TopicName [Partition]]
    ConsumerGroup => string
    TopicName => string
    Partition => int32
```

__init__(*consumer_group*, *partition_requests=[]*)
Create a new offset fetch request :param partition_requests: Iterable of

kafka.pykafka.protocol.PartitionOffsetFetchRequest for this request

__len__()
Length of the serialized message, in bytes

get_bytes()
Serialize the message :returns: Serialized message :rtype: bytearray

class pykafka.protocol.**OffsetFetchPartitionResponse**(*offset*, *metadata*, *err*)
Bases: tuple

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *offset*, *metadata*, *err*)
Create new instance of OffsetFetchPartitionResponse(offset, metadata, err)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0x906d60>*, *len=<built-in function len>*)
Make a new OffsetFetchPartitionResponse object from a sequence or iterable

_replace(***kwds*)
Return a new OffsetFetchPartitionResponse object replacing specified fields with new values

err
Alias for field number 2

metadata
Alias for field number 1

offset
Alias for field number 0

class pykafka.protocol.**OffsetFetchResponse**(*buff*)
Bases: pykafka.protocol.base.Response

An offset fetch response v0 Specification:: OffsetFetch Response (Version: 0) => [responses]

responses => topic [partition_responses] topic => STRING partition_responses => partition offset metadata error_code

partition => INT32 offset => INT64 metadata => NULLABLE_STRING error_code => INT16

__init__(*buff*)
Deserialize into a new Response :param buff: Serialized message :type buff: bytearray

class pykafka.protocol.**JoinGroupRequest**(*group_id*, *member_id*, *topic_name*, *membership_protocol*, *session_timeout=30000*)
Bases: pykafka.protocol.base.Request

A group join request Specification:: JoinGroupRequest => GroupId SessionTimeout MemberId ProtocolType GroupProtocols

GroupId => string SessionTimeout => int32 MemberId => string ProtocolType => string GroupProtocols => [ProtocolName ProtocolMetadata]

ProtocolName => string ProtocolMetadata => bytes

__init__(*group_id*, *member_id*, *topic_name*, *membership_protocol*, *session_timeout=30000*)
Create a new group join request

__len__()
Length of the serialized message, in bytes

get_bytes()
Serialize the message :returns: Serialized message :rtype: bytearray

class pykafka.protocol.**JoinGroupResponse**(*buff*)
Bases: pykafka.protocol.base.Response

A group join response Specification:: JoinGroupResponse => ErrorCode GenerationId GroupProtocol LeaderId MemberId Members

ErrorCode => int16 GenerationId => int32 GroupProtocol => string LeaderId => string MemberId => string Members => [MemberId MemberMetadata]

MemberId => string MemberMetadata => bytes

__init__(*buff*)
Deserialize into a new Response :param buff: Serialized message :type buff: bytearray

class pykafka.protocol.**SyncGroupRequest**(*group_id*, *generation_id*, *member_id*, *group_assignment*)
Bases: pykafka.protocol.base.Request

A group sync request Specification:: SyncGroupRequest => GroupId GenerationId MemberId GroupAssignment

GroupId => string GenerationId => int32 MemberId => string GroupAssignment => [MemberId MemberAssignment]

MemberId => string MemberAssignment => bytes

__init__(*group_id*, *generation_id*, *member_id*, *group_assignment*)
Create a new group join request

__len__()
Length of the serialized message, in bytes

get_bytes()
Serialize the message :returns: Serialized message :rtype: bytearray

class pykafka.protocol.**SyncGroupResponse**(*buff*)
Bases: pykafka.protocol.base.Response

A group sync response Specification:: SyncGroupResponse => ErrorCode MemberAssignment

ErrorCode => int16 MemberAssignment => bytes

__init__(*buff*)
Deserialize into a new Response :param buff: Serialized message :type buff: bytearray

class pykafka.protocol.**HeartbeatRequest**(*group_id*, *generation_id*, *member_id*)
Bases: pykafka.protocol.base.Request

A group heartbeat request Specification:: HeartbeatRequest => GroupId GenerationId MemberId

GroupId => string GenerationId => int32 MemberId => string

__init__(*group_id*, *generation_id*, *member_id*)
Create a new heartbeat request

__len__()
Length of the serialized message, in bytes

get_bytes()
Serialize the message :returns: Serialized message :rtype: bytearray

class pykafka.protocol.**HeartbeatResponse**(*buff*)
Bases: pykafka.protocol.base.Response

A group heartbeat response Specification:: HeartbeatResponse => ErrorCode

ErrorCode => int16

__init__(*buff*)
Deserialize into a new Response :param buff: Serialized message :type buff: bytearray

class pykafka.protocol.**LeaveGroupRequest**(*group_id*, *member_id*)
Bases: pykafka.protocol.base.Request

A group exit request Specification:: LeaveGroupRequest => GroupId MemberId

GroupId => string MemberId => string

__init__(*group_id*, *member_id*)
Create a new group join request

__len__()
Length of the serialized message, in bytes

get_bytes()
Serialize the message :returns: Serialized message :rtype: bytearray

class pykafka.protocol.**LeaveGroupResponse**(*buff*)
Bases: pykafka.protocol.base.Response

A group exit response Specification:: LeaveGroupResponse => ErrorCode

ErrorCode => int16

__init__(*buff*)
Deserialize into a new Response :param buff: Serialized message :type buff: bytearray

class pykafka.protocol.**ListGroupsRequest**
Bases: pykafka.protocol.base.Request

A list groups request

Specification:

ListGroupsRequest =>

__len__()
Length of the serialized message, in bytes

get_bytes()
Create a new list group request

class pykafka.protocol.**ListGroupsResponse**(*buff*)
Bases: pykafka.protocol.base.Response

A list groups response

Specification:

ListGroupsResponse => ErrorCode Groups ErrorCode => int16 Groups => [GroupId ProtocolType]

GroupId => string ProtocolType => string

__init__(*buff*)
Deserialize into a new Response

Parameters **buff** (bytearray) – Serialized message

class pykafka.protocol.**DescribeGroupsRequest**(*group_ids*)
Bases: pykafka.protocol.base.Request

A describe groups request

Specification:

DescribeGroupsRequest => [GroupId] GroupId => string

__init__(*group_ids*)
x.__init__(...) initializes x; see help(type(x)) for signature

__len__()
Length of the serialized message, in bytes

get_bytes()
Create a new list group request

class pykafka.protocol.**DescribeGroupsResponse**(*buff*)
Bases: pykafka.protocol.base.Response

A describe groups response

Specification:

DescribeGroupsResponse => [ErrorCode GroupId State ProtocolType Protocol Members] ErrorCode => int16 GroupId => string State => string ProtocolType => string Protocol => string Members => [MemberId ClientId ClientHost MemberMetadata MemberAssignment]

MemberId => string ClientId => string ClientHost => string MemberMetadata => bytes MemberAssignment => bytes

__init__(*buff*)
Deserialize into a new Response

Parameters **buff** (bytearray) – Serialized message

class pykafka.protocol.**Message**(*value*, *partition_key=None*, *compression_type=0*, *offset=-1*, *partition_id=-1*, *produce_attempt=0*, *protocol_version=0*, *timestamp=None*, *delivery_report_q=None*)
Bases: *pykafka.common.Message*, pykafka.utils.Serializable

Representation of a Kafka Message

NOTE: Compression is handled in the protocol because of the way Kafka embeds compressed MessageSets within Messages

Specification:

```
Message => Crc MagicByte Attributes Key Value
  Crc => int32
  MagicByte => int8
  Attributes => int8
  Key => bytes
  Value => bytes
```

pykafka.protocol.Message also contains *partition* and *partition_id* fields. Both of these have meaningless default values. When *pykafka.protocol.Message* is used by the producer, *partition_id* identifies the Message's destination partition. When used in a *pykafka.protocol.FetchRequest*, *partition_id* is set to the id of the partition from which the message was sent on receipt of the message. In the *pykafka.simpleconsumer.SimpleConsumer*, *partition* is set to the *pykafka.partition.Partition* instance from which the message was sent.

Variables

- **compression_type** – The compression algorithm used to generate the message's current value. Internal use only - regardless of the algorithm used, this will be *CompressionType.NONE* in any publicly accessible 'Message's.
- **partition_key** – Value used to assign this message to a particular partition.
- **value** – The payload associated with this message
- ***offset*** – The offset of the message
- **partition_id** – The id of the partition to which this message belongs
- **delivery_report_q** – For use by *pykafka.producer.Producer*

__init__(*value*, *partition_key=None*, *compression_type=0*, *offset=-1*, *partition_id=-1*, *produce_attempt=0*, *protocol_version=0*, *timestamp=None*, *delivery_report_q=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

__len__()
Length of the bytes that will be sent to the Kafka server.

pack_into(*buff*, *offset*)
Serialize and write to `buff` starting at offset `offset`.

Intentionally follows the pattern of `struct.pack_into`

Parameters

- **buff** – The buffer to write into
- **offset** – The offset to start the write at

timestamp_dt
Get the timestamp as a datetime, if valid

class pykafka.protocol.**MessageSet**(*compression_type=0*, *messages=None*, *broker_version='0.9.0'*)
Bases: `pykafka.utils.Serializable`

Representation of a set of messages in Kafka

This isn't useful outside of direct communications with Kafka, so we keep it hidden away here.

N.B.: MessageSets are not preceded by an int32 like other array elements in the protocol.

Specification:

```
MessageSet => [Offset MessageSize Message]
  Offset => int64
  MessageSize => int32
```

__init__(*compression_type=0*, *messages=None*, *broker_version='0.9.0'*)
Create a new MessageSet

Parameters

- **compression_type** – Compression to use on the messages
- **messages** – An initial list of messages for the set
- **broker_version** – A broker version with which this MessageSet is compatible

__len__()
Length of the serialized message, in bytes

We don't put the MessageSetSize in front of the serialization because that's *technically* not part of the MessageSet. Most requests/responses using MessageSets need that size, though, so be careful when using this.

__weakref__
list of weak references to the object (if defined)

_get_compressed()
Get a compressed representation of all current messages.

Returns a Message object with correct headers set and compressed data in the value field.

classmethod decode(*buff*, *partition_id=-1*)
Decode a serialized MessageSet.

pack_into(*buff*, *offset*)
Serialize and write to `buff` starting at offset `offset`.

Intentionally follows the pattern of `struct.pack_into`

Parameters

- **buff** – The buffer to write into

- **offset** – The offset to start the write at

class pykafka.protocol.**ApiVersionsRequest**
Bases: pykafka.protocol.base.Request

An api versions request

Specification:

```
ApiVersions Request (Version: 0) =>
```

__len__()
Length of the serialized message, in bytes

get_bytes()
Create a new api versions request

class pykafka.protocol.**ApiVersionsResponse**(*buff*)
Bases: pykafka.protocol.base.Response

Specification:

ApiVersions Response (Version: 0) => error_code [api_versions] error_code => INT16 api_versions => api_key min_version max_version

api_key => INT16 min_version => INT16 max_version => INT16

__init__(*buff*)
Deserialize into a new Response

Parameters **buff** (bytearray) – Serialized message

class pykafka.protocol.**CreateTopicsRequest**(*topic_requests*, *timeout=0*)
Bases: pykafka.protocol.base.Request

A create topics request

Specification:

CreateTopics Request (Version: 0) => [create_topic_requests] timeout

create_topic_requests => topic num_partitions replication_factor [replica_assignment] [config_entries] topic => STRING num_partitions => INT32 replication_factor => INT16 replica_assignment => partition [replicas]

partition => INT32 replicas => INT32

config_entries => config_name config_value config_name => STRING config_value => NULLABLE_STRING

timeout => INT32

__init__(*topic_requests*, *timeout=0*)
x.__init__(...) initializes x; see help(type(x)) for signature

__len__()
Length of the serialized message, in bytes

get_bytes()
Create a new create topics request

class pykafka.protocol.**CreateTopicsResponse**(*buff*)
Bases: pykafka.protocol.base.Response

A create topics response

Specification:

CreateTopics Response (Version: 0) => [topic_errors]

topic_errors => topic error_code topic => STRING error_code => INT16

__init__(*buff*)
Deserialize into a new Response

Parameters **buff** (bytearray) – Serialized message

class pykafka.protocol.**DeleteTopicsRequest**(*topics*, *timeout=0*)
Bases: pykafka.protocol.base.Request

A delete topics request

Specification:

DeleteTopics Request (Version: 0) => [topics] timeout topics => STRING timeout => INT32

__init__(*topics*, *timeout=0*)
x.__init__(...) initializes x; see help(type(x)) for signature

__len__()
Length of the serialized message, in bytes

get_bytes()
Create a new delete topics request

class pykafka.protocol.**DeleteTopicsResponse**(*buff*)
Bases: pykafka.protocol.base.Response

A delete topics response

Specification:

DeleteTopics Response (Version: 0) => [topic_error_codes]

topic_error_codes => topic error_code topic => STRING error_code => INT16

__init__(*buff*)
Deserialize into a new Response

Parameters **buff** (bytearray) – Serialized message

class pykafka.protocol.**PartitionOffsetRequest**
Bases: pykafka.protocol.offset.PartitionOffsetRequest

Offset request for a specific topic/partition :ivar topic_name: Name of the topic to look up :ivar partition_id: Id of the partition to look up :ivar offsets_before: Retrieve offset information for messages before

this timestamp (ms). -1 will retrieve the latest offsets and -2 will retrieve the earliest available offset. If -2,only 1 offset is returned

Variables **max_offsets** – How many offsets to return

class pykafka.protocol.**ConsumerGroupProtocolMetadata**(*version=0*, *topic_names=None*, *user_data='testuserdata'*)
Bases: object

Protocol specification:: ProtocolMetadata => Version Subscription UserData

Version => int16 Subscription => [Topic]

Topic => string

UserData => bytes

__init__ (*version=0*, *topic_names=None*, *user_data='testuserdata'*)
x.__init__(...) initializes x; see help(type(x)) for signature

__weakref__
list of weak references to the object (if defined)

class pykafka.protocol.**MemberAssignment** (*partition_assignment*, *version=1*)
Bases: object

Protocol specification:: MemberAssignment => Version PartitionAssignment

Version => int16 PartitionAssignment => [Topic [Partition]]

Topic => string Partition => int32

UserData => bytes

__init__ (*partition_assignment*, *version=1*)
x.__init__(...) initializes x; see help(type(x)) for signature

__weakref__
list of weak references to the object (if defined)

class pykafka.protocol.**FetchResponseV1** (*buff*, *offset=0*, *broker_version='0.9.0'*)
Bases: pykafka.protocol.fetch.FetchResponse

__init__ (*buff*, *offset=0*, *broker_version='0.9.0'*)
Deserialize into a new Response

Parameters

- **buff** (bytearray) – Serialized message
- **offset** (*int*) – Offset into the message

class pykafka.protocol.**FetchResponseV2** (*buff*, *offset=0*, *broker_version='0.9.0'*)
Bases: pykafka.protocol.fetch.FetchResponseV1

class pykafka.protocol.**MetadataResponseV1** (*buff*)
Bases: pykafka.protocol.metadata.MetadataResponse

Response from MetadataRequest Specification:: Metadata Response (Version: 1) => [brokers] controller_id [topic_metadata]

brokers => node_id host port rack node_id => INT32 host => STRING port => INT32 rack => NULLABLE_STRING (new since v0)

controller_id => INT32 (new since v0) topic_metadata => error_code topic is_internal [partition_metadata]

error_code => INT16 topic => STRING is_internal => BOOLEAN (new since v0) partition_metadata => error_code partition leader [replicas] [isr]

error_code => INT16 partition => INT32 leader => INT32 replicas => INT32 isr => INT32

__init__ (*buff*)
Deserialize into a new Response :param buff: Serialized message :type buff: bytearray

class pykafka.protocol.**MetadataRequestV1** (*topics=None*, **kwargs*)
Bases: pykafka.protocol.metadata.MetadataRequest

class pykafka.protocol.**CreateTopicRequest**
Bases: pykafka.protocol.admin.CreateTopicRequest

class pykafka.protocol.**ProducePartitionResponse**(*err*, *offset*)
Bases: tuple

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *err*, *offset*)
Create new instance of ProducePartitionResponse(err, offset)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0x906d60>*, *len=<built-in function len>*)
Make a new ProducePartitionResponse object from a sequence or iterable

_replace(***kwds*)
Return a new ProducePartitionResponse object replacing specified fields with new values

err
Alias for field number 0

offset
Alias for field number 1

class pykafka.protocol.**ListOffsetRequestV1**(*partition_requests*)
Bases: pykafka.protocol.offset.ListOffsetRequest

Specification::

ListOffsetRequest => ReplicaId [TopicName [Partition Time]] ReplicaId => int32 TopicName => string Partition => int32 Time => int64

__init__(*partition_requests*)
Create a new offset request

__len__()
Length of the serialized message, in bytes

get_bytes()
Serialize the message :returns: Serialized message :rtype: bytearray

class pykafka.protocol.**ListOffsetResponseV1**(*buff*)
Bases: pykafka.protocol.offset.ListOffsetResponse

Specification::

ListOffsetResponse => [TopicName [PartitionOffsets]] PartitionOffsets => Partition ErrorCode Timestamp [Offset] Partition => int32 ErrorCode => int16 Timestamp => int64 Offset => int64

__init__(*buff*)
Deserialize into a new Response :param buff: Serialized message :type buff: bytearray

class pykafka.protocol.**OffsetFetchRequestV1**(*consumer_group*, *partition_requests=[]*)
Bases: pykafka.protocol.offset_commit.OffsetFetchRequest

class pykafka.protocol.**OffsetFetchResponseV1**(*buff*)
Bases: pykafka.protocol.offset_commit.OffsetFetchResponse

An offset fetch response v1 (all the same as v0) Specification:: OffsetFetch Response (Version: 1) => [responses]

responses => topic [partition_responses] topic => STRING partition_responses => partition offset metadata error_code

partition => INT32 offset => INT64 metadata => NULLABLE_STRING error_code => INT16

class pykafka.protocol.**OffsetFetchRequestV2**(*consumer_group*, *partition_requests=[]*)
Bases: pykafka.protocol.offset_commit.OffsetFetchRequestV1

class pykafka.protocol.**OffsetFetchResponseV2**(*buff*)
Bases: pykafka.protocol.offset_commit.OffsetFetchResponseV1

An offset fetch response v2 Specification:: OffsetFetch Response (Version: 2) => [responses] error_code

responses => topic [partition_responses] topic => STRING partition_responses => partition offset metadata error_code

partition => INT32 offset => INT64 metadata => NULLABLE_STRING error_code => INT16

error_code => INT16 (new since v1)

__init__(*buff*)
Deserialize into a new Response :param buff: Serialized message :type buff: bytearray

class pykafka.protocol.**MetadataRequestV2**(*topics=None*, **kwargs*)
Bases: pykafka.protocol.metadata.MetadataRequestV1

class pykafka.protocol.**MetadataResponseV2**(*buff*)
Bases: pykafka.protocol.metadata.MetadataResponseV1

Response from MetadataRequest Specification:: Metadata Response (Version: 2) => [brokers] cluster_id controller_id [topic_metadata]

brokers => node_id host port rack node_id => INT32 host => STRING port => INT32 rack => NULLABLE_STRING

cluster_id => NULLABLE_STRING (new since v1) controller_id => INT32 topic_metadata => error_code topic is_internal [partition_metadata]

error_code => INT16 topic => STRING is_internal => BOOLEAN partition_metadata => error_code partition leader [replicas] [isr]

error_code => INT16 partition => INT32 leader => INT32 replicas => INT32 isr => INT32

__init__(*buff*)
Deserialize into a new Response :param buff: Serialized message :type buff: bytearray

class pykafka.protocol.**MetadataRequestV3**(*topics=None*, **kwargs*)
Bases: pykafka.protocol.metadata.MetadataRequestV2

class pykafka.protocol.**MetadataResponseV3**(*buff*)
Bases: pykafka.protocol.metadata.MetadataResponseV2

Response from MetadataRequest Specification:: Metadata Response (Version: 3) => throttle_time_ms [brokers] cluster_id controller_id [topic_metadata]

throttle_time_ms => INT32 (new since v2) brokers => node_id host port rack

node_id => INT32 host => STRING port => INT32 rack => NULLABLE_STRING

cluster_id => NULLABLE_STRING controller_id => INT32 topic_metadata => error_code topic is_internal [partition_metadata]

error_code => INT16 topic => STRING is_internal => BOOLEAN partition_metadata => error_code partition leader [replicas] [isr]

error_code => INT16 partition => INT32 leader => INT32 replicas => INT32 isr => INT32

__init__(*buff*)
Deserialize into a new Response :param buff: Serialized message :type buff: bytearray

class pykafka.protocol.**MetadataRequestV4**(*topics=None*, *allow_topic_autocreation=True*)
Bases: pykafka.protocol.metadata.MetadataRequestV3

Metadata Request Specification:: Metadata Request (Version: 4) => [topics] allow_auto_topic_creation

topics => STRING allow_auto_topic_creation => BOOLEAN

__init__(*topics=None*, *allow_topic_autocreation=True*)
Create a new MetadataRequest :param topics: Topics to query. Leave empty for all available topics. :param allow_topic_autocreation: If this and the broker config

'auto.create.topics.enable' are true, topics that don't exist will be created by the broker. Otherwise, no topics will be created by the broker.

__len__()
Length of the serialized message, in bytes

get_bytes()
Serialize the message :returns: Serialized message :rtype: bytearray

class pykafka.protocol.**MetadataResponseV4**(*buff*)
Bases: pykafka.protocol.metadata.MetadataResponseV3

Response from MetadataRequest Specification:: Metadata Response (Version: 4) => throttle_time_ms [brokers] cluster_id controller_id [topic_metadata]

throttle_time_ms => INT32 brokers => node_id host port rack

node_id => INT32 host => STRING port => INT32 rack => NULLABLE_STRING

cluster_id => NULLABLE_STRING controller_id => INT32 topic_metadata => error_code topic is_internal [partition_metadata]

error_code => INT16 topic => STRING is_internal => BOOLEAN partition_metadata => error_code partition leader [replicas] [isr]

error_code => INT16 partition => INT32 leader => INT32 replicas => INT32 isr => INT32

class pykafka.protocol.**MetadataRequestV5**(*topics=None*, *allow_topic_autocreation=True*)
Bases: pykafka.protocol.metadata.MetadataRequestV4

class pykafka.protocol.**MetadataResponseV5**(*buff*)
Bases: pykafka.protocol.metadata.MetadataResponseV4

Response from MetadataRequest Specification:: Metadata Response (Version: 5) => throttle_time_ms [brokers] cluster_id controller_id [topic_metadata]

throttle_time_ms => INT32 brokers => node_id host port rack

node_id => INT32 host => STRING port => INT32 rack => NULLABLE_STRING

cluster_id => NULLABLE_STRING controller_id => INT32 topic_metadata => error_code topic is_internal [partition_metadata]

error_code => INT16 topic => STRING is_internal => BOOLEAN partition_metadata => error_code partition leader [replicas] [isr] [offline_replicas]

error_code => INT16 partition => INT32 leader => INT32 replicas => INT32 isr => INT32 offline_replicas => INT32 (new since v4)

__init__(*buff*)

Deserialize into a new Response :param buff: Serialized message :type buff: `bytearray`

6.2.15 pykafka.simpleconsumer

class pykafka.simpleconsumer.**SimpleConsumer**(*topic*, *cluster*, *consumer_group=None*, *partitions=None*, *fetch_message_max_bytes=1048576*, *num_consumer_fetchers=1*, *auto_commit_enable=False*, *auto_commit_interval_ms=60000*, *queued_max_messages=2000*, *fetch_min_bytes=1*, *fetch_error_backoff_ms=500*, *fetch_wait_max_ms=100*, *offsets_channel_backoff_ms=1000*, *offsets_commit_max_retries=5*, *auto_offset_reset=-2*, *consumer_timeout_ms=-1*, *auto_start=True*, *reset_offset_on_start=False*, *compacted_topic=False*, *generation_id=-1*, *consumer_id=''*, *deserializer=None*, *reset_offset_on_fetch=True*)

Bases: `object`

A non-balancing consumer for Kafka

__del__()

Stop consumption and workers when object is deleted

__init__(*topic*, *cluster*, *consumer_group=None*, *partitions=None*, *fetch_message_max_bytes=1048576*, *num_consumer_fetchers=1*, *auto_commit_enable=False*, *auto_commit_interval_ms=60000*, *queued_max_messages=2000*, *fetch_min_bytes=1*, *fetch_error_backoff_ms=500*, *fetch_wait_max_ms=100*, *offsets_channel_backoff_ms=1000*, *offsets_commit_max_retries=5*, *auto_offset_reset=-2*, *consumer_timeout_ms=-1*, *auto_start=True*, *reset_offset_on_start=False*, *compacted_topic=False*, *generation_id=-1*, *consumer_id=''*, *deserializer=None*, *reset_offset_on_fetch=True*)

Create a SimpleConsumer.

Settings and default values are taken from the Scala consumer implementation. Consumer group is included because it's necessary for offset management, but doesn't imply that this is a balancing consumer. Use a BalancedConsumer for that.

Parameters

- **topic** (*pykafka.topic.Topic*) – The topic this consumer should consume

- **cluster** (*pykafka.cluster.Cluster*) – The cluster to which this consumer should connect
- **consumer_group** (*str*) – The name of the consumer group this consumer should use for offset committing and fetching.
- **partitions** (Iterable of *pykafka.partition.Partition*) – Existing partitions to which to connect
- **fetch_message_max_bytes** (*int*) – The number of bytes of messages to attempt to fetch
- **num_consumer_fetchers** (*int*) – The number of workers used to make FetchRequests
- **auto_commit_enable** (*bool*) – If true, periodically commit to kafka the offset of messages already returned from consume() calls. Requires that *consumer_group* is not *None*.
- **auto_commit_interval_ms** (*int*) – The frequency (in milliseconds) at which the consumer offsets are committed to kafka. This setting is ignored if *auto_commit_enable* is *False*.
- **queued_max_messages** (*int*) – Maximum number of messages buffered for consumption per partition
- **fetch_min_bytes** (*int*) – The minimum amount of data (in bytes) the server should return for a fetch request. If insufficient data is available the request will block until sufficient data is available.
- **fetch_error_backoff_ms** (*int*) – The amount of time (in milliseconds) that the consumer should wait before retrying after an error. Errors include absence of data (*RD_KAFKA_RESP_ERR__PARTITION_EOF*), so this can slow a normal fetch scenario. Only used by the native consumer (*RdKafkaSimpleConsumer*).
- **fetch_wait_max_ms** (*int*) – The maximum amount of time (in milliseconds) the server will block before answering the fetch request if there isn't sufficient data to immediately satisfy *fetch_min_bytes*.
- **offsets_channel_backoff_ms** (*int*) – Backoff time (in milliseconds) to retry offset commits/fetches
- **offsets_commit_max_retries** (*int*) – Retry the offset commit up to this many times on failure.
- **auto_offset_reset** (*pykafka.common.OffsetType*) – What to do if an offset is out of range. This setting indicates how to reset the consumer's internal offset counter when an *OffsetOutOfRangeError* is encountered.
- **consumer_timeout_ms** (*int*) – Amount of time (in milliseconds) the consumer may spend without messages available for consumption before returning None.
- **auto_start** (*bool*) – Whether the consumer should begin communicating with kafka after __init__ is complete. If false, communication can be started with *start()*.
- **reset_offset_on_start** (*bool*) – Whether the consumer should reset its internal offset counter to *self._auto_offset_reset* and commit that offset immediately upon starting up
- **compacted_topic** (*bool*) – Set to read from a compacted topic. Forces consumer to use less stringent message ordering logic because compacted topics do not provide offsets in strict incrementing order.

- **generation_id** (*int*) – Deprecated::2.7 Do not set if directly instantiating SimpleConsumer. The generation id with which to make group requests
- **consumer_id** (*bytes*) – Deprecated::2.7 Do not set if directly instantiating SimpleConsumer. The identifying string to use for this consumer on group requests
- **deserializer** (*function*) – A function defining how to deserialize messages returned from Kafka. A function with the signature d(value, partition_key) that returns a tuple of (deserialized_value, deserialized_partition_key). The arguments passed to this function are the bytes representations of a message's value and partition key, and the returned data should be these fields transformed according to the client code's serialization logic. See *pykafka.utils.__init__* for stock implemtations.
- **reset_offset_on_fetch** (*bool*) – Whether to update offsets during fetch_offsets. Disable for read-only use cases to prevent side-effects.

__iter__()
Yield an infinite stream of messages until the consumer times out

__repr__(*)* <==> *repr(x)*

__weakref__
list of weak references to the object (if defined)

_auto_commit()
Commit offsets only if it's time to do so

_build_default_error_handlers()
Set up the error handlers to use for partition errors.

_discover_group_coordinator()
Set the group coordinator for this consumer.

If a consumer group is not supplied to __init__, this method does nothing

_raise_worker_exceptions()
Raises exceptions encountered on worker threads

_setup_autocommit_worker()
Start the autocommitter thread

_setup_fetch_workers()
Start the fetcher threads

_update()
Update the consumer and cluster after an ERROR_CODE

_wait_for_slot_available()
Block until at least one queue has less than *_queued_max_messages*

commit_offsets(*partition_offsets=None*)
Commit offsets for this consumer's partitions

Uses the offset commit/fetch API

Parameters **partition_offsets** (Sequence of tuples of the form (*pykafka.partition.Partition*, int)) – (*partition*, *offset*) pairs to commit where *partition* is the partition for which to commit the offset and *offset* is the offset to commit for the partition. Note that using this argument when *auto_commit_enable* is enabled can cause inconsistencies in committed offsets. For best results, use *either* this argument *or* *auto_commit_enable*.

consume(*block=True*, *unblock_event=None*)
Get one message from the consumer.

Parameters

- **block** (*bool*) – Whether to block while waiting for a message
- **unblock_event** (threading.Event) – Return when the event is set()

fetch()
Fetch new messages for all partitions

Create a FetchRequest for each broker and send it. Enqueue each of the returned messages in the approprate OwnedPartition.

fetch_offsets()
Fetch offsets for this consumer's topic

Uses the offset commit/fetch API

Returns List of (id, *pykafka.protocol.OffsetFetchPartitionResponse*) tuples

held_offsets
Return a map from partition id to held offset for each partition

partitions
A list of the partitions that this consumer consumes

reset_offsets(*partition_offsets=None*)
Reset offsets for the specified partitions

For each value provided in *partition_offsets*: if the value is an integer, immediately reset the partition's internal offset counter to that value. If it's a *datetime.datetime* instance or a valid *OffsetType*, issue a *ListOffsetRequest* using that timestamp value to discover the latest offset in the latest log segment before that timestamp, then set the partition's internal counter to that value.

Parameters **partition_offsets** (Sequence of tuples of the form (*pykafka.partition.Partition*, int OR *datetime.datetime*)) – (*partition*, *timestamp_or_offset*) pairs to reset where *partition* is the partition for which to reset the offset and *timestamp_or_offset* is EITHER the timestamp before which to find a valid offset to set the partition's counter to OR the new offset the partition's counter should be set to.

start()
Begin communicating with Kafka, including setting up worker threads

Fetches offsets, starts an offset autocommitter worker pool, and starts a message fetcher worker pool.

stop()
Flag all running workers for deletion.

topic
The topic this consumer consumes

6.2.16 pykafka.topic

Author: Keith Bourgoin, Emmett Butler

class pykafka.topic.**Topic**(*cluster*, *topic_metadata*)
Bases: object

A Topic is an abstraction over the kafka concept of a topic. It contains a dictionary of partitions that comprise it.

__init__(*cluster*, *topic_metadata*)
Create the Topic from metadata.

Parameters

- **cluster** (*pykafka.cluster.Cluster*) – The Cluster to use
- **topic_metadata** (pykafka.protocol.TopicMetadata) – Metadata for all topics.

__repr__(*) <==> *repr(x)*

__weakref__
list of weak references to the object (if defined)

earliest_available_offsets()
Get the earliest offset for each partition of this topic.

fetch_offset_limits(*offsets_before*, *max_offsets=1*)
Get information about the offsets of log segments for this topic

The ListOffsets API, which this function relies on, primarily deals with topics in terms of their log segments. Its behavior can be summed up as follows: it returns some subset of starting message offsets for the log segments of each partition. The particular subset depends on this function's two arguments, filtering by timestamp and in certain cases, count. The documentation for this API is notoriously imprecise, so here's a little example to illustrate how it works.

Take a topic with three partitions 0,1,2. 2665 messages have been produced to this topic, and the brokers' *log.segment.bytes* settings are configured such that each log segment contains roughly 530 messages. The two oldest log segments have been deleted due to log retention settings such as *log.retention.hours*. Thus, the *log.dirs* currently contains these files for partition 0:

/var/local/kafka/data/test2-0/00000000000000001059.log /var/local/kafka/data/test2-0/00000000000000002119.log /var/local/kafka/data/test2-0/00000000000000001589.log /var/local/kafka/data/test2-0/00000000000000002649.log

The numbers on these filenames indicate the offset of the earliest message contained within. The most recent message was written at 1523572215.69.

Given this log state, a call to this function with offsets_before=OffsetType.LATEST and max_offsets=100 will result in a return value of [2665,2649,2119,1589,1059] for partition 0. The first value (2665) is the offset of the latest available message from the latest log segment. The other four offsets are those of the earliest messages from each log segment for the partition. Changing max_offsets to 3 will result in only the first three elements of this list being returned.

A call to this function with offsets_before=OffsetType.EARLIEST will result in a value of [1059] - only the offset of the earliest message present in log segments for partition 0. In this case, the return value is not affected by max_offsets.

A call to this function with offsets_before=(1523572215.69 * 1000) (the timestamp in milliseconds of the very last message written to the partition) will result in a value of [2649,2119,1589,1059]. This is the same list as with OffsetType.LATEST, but with the first element removed. This is because unlike the other elements, the message with this offset (2665) was not written *before* the given timestamp.

In cases where there are no log segments fitting the given criteria for a partition, an empty list is returned. This applies if the given timestamp is before the write time of the oldest message in the partition, as well as if there are no log segments for the partition.

Thanks to Andras Beni from the Kafka users mailing list for providing this example.

Parameters

- **offsets_before** (*datetime.datetime* or int) – Epoch timestamp in milliseconds or datetime indicating the latest write time for returned offsets. Only offsets of messages written before this timestamp will be returned. Permissible special values are *common.OffsetType.LATEST*, indicating that offsets from all available log segments should be returned, and *common.OffsetType.EARLIEST*, indicating that only the offset of the earliest available message should be returned. Deprecated::2.7,3.6: do not use int
- **max_offsets** (*int*) – The maximum number of offsets to return when more than one is available. In the case where *offsets_before* == *OffsetType.EARLIEST*, this parameter is meaningless since there is always either one or zero earliest offsets. In other cases, this parameter slices off the earliest end of the list, leaving the latest *max_offsets* offsets.

get_balanced_consumer(*consumer_group*, *managed=False*, ***kwargs*)
Return a BalancedConsumer of this topic

Parameters

- **consumer_group** (*bytes*) – The name of the consumer group to join
- **managed** (*bool*) – If True, manage the consumer group with Kafka using the 0.9 group management api (requires Kafka >=0.9))

get_producer(*use_rdkafka=False*, ***kwargs*)
Create a *pykafka.producer.Producer* for this topic.

For a description of all available *kwargs*, see the Producer docstring.

get_simple_consumer(*consumer_group=None*, *use_rdkafka=False*, ***kwargs*)
Return a SimpleConsumer of this topic

Parameters

- **consumer_group** (*bytes*) – The name of the consumer group to join
- **use_rdkafka** (*bool*) – Use librdkafka-backed consumer if available

get_sync_producer(***kwargs*)
Create a *pykafka.producer.Producer* for this topic.

The created *Producer* instance will have *sync=True*.

For a description of all available *kwargs*, see the Producer docstring.

latest_available_offsets()
Fetch the next available offset

Get the offset of the next message that would be appended to each partition of this topic.

name
The name of this topic

partitions
A dictionary containing all known partitions for this topic

update(*metadata*)
Update the Partitions with metadata about the cluster.

Parameters **metadata** (pykafka.protocol.TopicMetadata) – Metadata for all topics

6.2.17 pykafka.utils.compression

Author: Keith Bourgoin

pykafka.utils.compression.**encode_gzip**(*buff*)
Encode a buffer using gzip

pykafka.utils.compression.**decode_gzip**(*buff*)
Decode a buffer using gzip

pykafka.utils.compression.**encode_snappy**(*buff*, *xerial_compatible=False*, *xerial_blocksize=32768*)
Encode a buffer using snappy

If xerial_compatible is set, the buffer is encoded in a fashion compatible with the xerial snappy library.

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

The format is as follows: +————-+————+————+————+————+ | Header | Block1 len | Block1 data | Blockn len | Blockn data | |————-+————+————+————+————| | 16 bytes | BE int32 | snappy bytes | BE int32 | snappy bytes | +————-+————+————+————+————+

It is important to note that *blocksize* is the amount of uncompressed data presented to snappy at each block, whereas *blocklen* is the number of bytes that will be present in the stream.

Adapted from kafka-python https://github.com/mumrah/kafka-python/pull/127/files

pykafka.utils.compression.**decode_snappy**(*buff*)
Decode a buffer using Snappy

If xerial is found to be in use, the buffer is decoded in a fashion compatible with the xerial snappy library.

Adapted from kafka-python https://github.com/mumrah/kafka-python/pull/127/files

pykafka.utils.compression.**encode_lz4_old_kafka**(*buff*)
Encode buff for 0.8/0.9 brokers – requires an incorrect header checksum.

Reference impl: https://github.com/dpkp/kafka-python/blob/a00f9ead161e8b05ac953b460950e42fa0e0b7d6/kafka/codec.py#L227

pykafka.utils.compression.**decode_lz4_old_kafka**(*buff*)
Decode buff for 0.8/0.9 brokers

Reference impl: https://github.com/dpkp/kafka-python/blob/a00f9ead161e8b05ac953b460950e42fa0e0b7d6/kafka/codec.py#L258

6.2.18 pykafka.utils.error_handlers

Author: Emmett Butler

pykafka.utils.error_handlers.**handle_partition_responses**(*error_handlers*, *parts_by_error=None*, *success_handler=None*, *response=None*, *partitions_by_id=None*)
Call the appropriate handler for each errored partition

Parameters

- **error_handlers** (*dict {int: callable(parts)}*) – mapping of error code to handler
- **parts_by_error** (dict {int: iterable(`pykafka.simpleconsumer.OwnedPartition`)}) – a dict of partitions grouped by error code

- **success_handler** (*callable accepting an iterable of partition responses*) – function to call for successful partitions
- **response** (pykafka.protocol.Response) – a Response object containing partition responses
- **partitions_by_id** (dict {int: pykafka.simpleconsumer.OwnedPartition}) – a dict mapping partition ids to OwnedPartition instances

pykafka.utils.error_handlers.**raise_error**(*error*, *info=''*)
Raise the given error

6.2.19 pykafka.utils.socket

Author: Keith Bourgoin, Emmett Butler

pykafka.utils.socket.**recvall_into**(*socket*, *bytea*, *size*)
Reads *size* bytes from the socket into the provided bytearray (modifies in-place.)

This is basically a hack around the fact that *socket.recv_into* doesn't allow buffer offsets.

Return type *bytearray*

6.2.20 pykafka.utils.struct_helpers

Author: Keith Bourgoin, Emmett Butler

pykafka.utils.struct_helpers.**unpack_from**(*fmt*, *buff*, *offset=0*)
A customized version of *struct.unpack_from*

This is a conveinence function that makes decoding the arrays, strings, and byte arrays that we get from Kafka significantly easier. It takes the same arguments as *struct.unpack_from* but adds 3 new formats:

- Wrap a section in *[]* to indicate an array. e.g.: *[ii]*
- *S* for strings (int16 followed by byte array)
- *Y* for byte arrays (int32 followed by byte array)

Spaces are ignored in the format string, allowing more readable formats

NOTE: This may be a performance bottleneck. We're avoiding a lot of memory allocations by using the same buffer, but if we could call *struct.unpack_from* only once, that's about an order of magnitude faster. However, constructing the format string to do so would erase any gains we got from having the single call.

6.3 Indices and tables

- genindex
- modindex
- search

Python Module Index

p

pykafka.balancedconsumer, 16
pykafka.broker, 20
pykafka.client, 24
pykafka.cluster, 26
pykafka.common, 28
pykafka.connection, 28
pykafka.exceptions, 30
pykafka.handlers, 33
pykafka.managedbalancedconsumer, 38
pykafka.membershipprotocol, 41
pykafka.partition, 43
pykafka.partitioners, 44
pykafka.producer, 45
pykafka.protocol, 49
pykafka.simpleconsumer, 67
pykafka.topic, 70
pykafka.utils.compression, 72
pykafka.utils.error_handlers, 73
pykafka.utils.socket, 74
pykafka.utils.struct_helpers, 74

Index

Symbols

__call__() (pykafka.partitioners.BasePartitioner method), 44
__call__() (pykafka.partitioners.GroupHashingPartitioner method), 45
__call__() (pykafka.partitioners.HashingPartitioner method), 44
__call__() (pykafka.partitioners.RandomPartitioner method), 44
__del__() (pykafka.connection.BrokerConnection method), 29
__del__() (pykafka.simpleconsumer.SimpleConsumer method), 67
__enter__() (pykafka.handlers.ThreadingHandler.Semaphore method), 35
__enter__() (pykafka.producer.Producer method), 46
__eq__() (pykafka.partition.Partition method), 43
__exit__() (pykafka.producer.Producer method), 46
__getnewargs__() (pykafka.handlers.RequestHandler.Shared method), 36
__getnewargs__() (pykafka.handlers.RequestHandler.Task method), 37
__getnewargs__() (pykafka.membershipprotocol.GroupMembershipProtocol method), 41
__getnewargs__() (pykafka.protocol.FetchPartitionResponse method), 51
__getnewargs__() (pykafka.protocol.OffsetCommitPartitionResponse method), 54
__getnewargs__() (pykafka.protocol.OffsetFetchPartitionResponse method), 55
__getnewargs__() (pykafka.protocol.ProducePartitionResponse method), 64
__getstate__() (pykafka.handlers.RequestHandler.Shared method), 36
__getstate__() (pykafka.handlers.RequestHandler.Task method), 37
__getstate__() (pykafka.membershipprotocol.GroupMembershipProtocol method), 41
__getstate__() (pykafka.protocol.FetchPartitionResponse method), 52
__getstate__() (pykafka.protocol.OffsetCommitPartitionResponse method), 54
__getstate__() (pykafka.protocol.OffsetFetchPartitionResponse method), 55
__getstate__() (pykafka.protocol.ProducePartitionResponse method), 64
__hash__() (pykafka.partition.Partition method), 43
__init__() (pykafka.balancedconsumer.BalancedConsumer method), 16
__init__() (pykafka.broker.Broker method), 21
__init__() (pykafka.client.KafkaClient method), 25
__init__() (pykafka.cluster.Cluster method), 26
__init__() (pykafka.connection.BrokerConnection method), 29
__init__() (pykafka.connection.SslConfig method), 29
__init__() (pykafka.exceptions.PartitionOwnedError method), 32
__init__() (pykafka.handlers.RequestHandler method), 37
__init__() (pykafka.handlers.ResponseFuture method), 33
__init__() (pykafka.handlers.ThreadingHandler.Semaphore method), 36
__init__() (pykafka.managedbalancedconsumer.ManagedBalancedConsume method), 39
__init__() (pykafka.partition.Partition method), 43
__init__() (pykafka.partitioners.GroupHashingPartitioner method), 45
__init__() (pykafka.partitioners.HashingPartitioner method), 45
__init__() (pykafka.partitioners.RandomPartitioner method), 44
__init__() (pykafka.producer.Producer method), 46
__init__() (pykafka.protocol.ApiVersionsResponse method), 61
__init__() (pykafka.protocol.ConsumerGroupProtocolMetadata method), 63
__init__() (pykafka.protocol.CreateTopicsRequest method), 61

__init__() (pykafka.protocol.CreateTopicsResponse method), 62
__init__() (pykafka.protocol.DeleteTopicsRequest method), 62
__init__() (pykafka.protocol.DeleteTopicsResponse method), 62
__init__() (pykafka.protocol.DescribeGroupsRequest method), 58
__init__() (pykafka.protocol.DescribeGroupsResponse method), 59
__init__() (pykafka.protocol.FetchRequest method), 51
__init__() (pykafka.protocol.FetchResponse method), 52
__init__() (pykafka.protocol.FetchResponseV1 method), 63
__init__() (pykafka.protocol.GroupCoordinatorRequest method), 53
__init__() (pykafka.protocol.GroupCoordinatorResponse method), 53
__init__() (pykafka.protocol.HeartbeatRequest method), 57
__init__() (pykafka.protocol.HeartbeatResponse method), 57
__init__() (pykafka.protocol.JoinGroupRequest method), 56
__init__() (pykafka.protocol.JoinGroupResponse method), 56
__init__() (pykafka.protocol.LeaveGroupRequest method), 57
__init__() (pykafka.protocol.LeaveGroupResponse method), 58
__init__() (pykafka.protocol.ListGroupsResponse method), 58
__init__() (pykafka.protocol.ListOffsetRequest method), 53
__init__() (pykafka.protocol.ListOffsetRequestV1 method), 64
__init__() (pykafka.protocol.ListOffsetResponse method), 53
__init__() (pykafka.protocol.ListOffsetResponseV1 method), 64
__init__() (pykafka.protocol.MemberAssignment method), 63
__init__() (pykafka.protocol.Message method), 59
__init__() (pykafka.protocol.MessageSet method), 60
__init__() (pykafka.protocol.MetadataRequest method), 49
__init__() (pykafka.protocol.MetadataRequestV4 method), 66
__init__() (pykafka.protocol.MetadataResponse method), 49
__init__() (pykafka.protocol.MetadataResponseV1 method), 63
__init__() (pykafka.protocol.MetadataResponseV2 method), 65
__init__() (pykafka.protocol.MetadataResponseV3 method), 66
__init__() (pykafka.protocol.MetadataResponseV5 method), 67
__init__() (pykafka.protocol.OffsetCommitRequest method), 54
__init__() (pykafka.protocol.OffsetCommitResponse method), 55
__init__() (pykafka.protocol.OffsetFetchRequest method), 55
__init__() (pykafka.protocol.OffsetFetchResponse method), 56
__init__() (pykafka.protocol.OffsetFetchResponseV2 method), 65
__init__() (pykafka.protocol.ProduceRequest method), 49
__init__() (pykafka.protocol.ProduceResponse method), 50
__init__() (pykafka.protocol.SyncGroupRequest method), 57
__init__() (pykafka.protocol.SyncGroupResponse method), 57
__init__() (pykafka.simpleconsumer.SimpleConsumer method), 67
__init__() (pykafka.topic.Topic method), 70
__iter__() (pykafka.balancedconsumer.BalancedConsumer method), 18
__iter__() (pykafka.simpleconsumer.SimpleConsumer method), 69
__len__() (pykafka.protocol.ApiVersionsRequest method), 61
__len__() (pykafka.protocol.CreateTopicsRequest method), 61
__len__() (pykafka.protocol.DeleteTopicsRequest method), 62
__len__() (pykafka.protocol.DescribeGroupsRequest method), 58
__len__() (pykafka.protocol.FetchRequest method), 51
__len__() (pykafka.protocol.GroupCoordinatorRequest method), 53
__len__() (pykafka.protocol.HeartbeatRequest method), 57
__len__() (pykafka.protocol.JoinGroupRequest method), 56
__len__() (pykafka.protocol.LeaveGroupRequest method), 57
__len__() (pykafka.protocol.ListGroupsRequest method), 58
__len__() (pykafka.protocol.ListOffsetRequest method), 53
__len__() (pykafka.protocol.ListOffsetRequestV1 method), 64
__len__() (pykafka.protocol.Message method), 59
__len__() (pykafka.protocol.MessageSet method), 60

__len__() (pykafka.protocol.MetadataRequest method), 49
__len__() (pykafka.protocol.MetadataRequestV4 method), 66
__len__() (pykafka.protocol.OffsetCommitRequest method), 54
__len__() (pykafka.protocol.OffsetFetchRequest method), 55
__len__() (pykafka.protocol.ProduceRequest method), 50
__len__() (pykafka.protocol.SyncGroupRequest method), 57
__lt__() (pykafka.partition.Partition method), 43
__ne__() (pykafka.partition.Partition method), 43
__new__() (pykafka.handlers.RequestHandler.Shared static method), 36
__new__() (pykafka.handlers.RequestHandler.Task static method), 37
__new__() (pykafka.membershipprotocol.GroupMembershipProtocol static method), 41
__new__() (pykafka.protocol.FetchPartitionResponse static method), 52
__new__() (pykafka.protocol.OffsetCommitPartitionResponse static method), 54
__new__() (pykafka.protocol.OffsetFetchPartitionResponse static method), 55
__new__() (pykafka.protocol.ProducePartitionResponse static method), 64
__repr__() (pykafka.balancedconsumer.BalancedConsumer method), 18
__repr__() (pykafka.broker.Broker method), 21
__repr__() (pykafka.client.KafkaClient method), 25
__repr__() (pykafka.cluster.Cluster method), 26
__repr__() (pykafka.handlers.RequestHandler.Shared method), 36
__repr__() (pykafka.handlers.RequestHandler.Task method), 37
__repr__() (pykafka.membershipprotocol.GroupMembershipProtocol method), 41
__repr__() (pykafka.partition.Partition method), 43
__repr__() (pykafka.producer.Producer method), 47
__repr__() (pykafka.protocol.FetchPartitionResponse method), 52
__repr__() (pykafka.protocol.OffsetCommitPartitionResponse method), 54
__repr__() (pykafka.protocol.OffsetFetchPartitionResponse method), 55
__repr__() (pykafka.protocol.ProducePartitionResponse method), 64
__repr__() (pykafka.simpleconsumer.SimpleConsumer method), 69
__repr__() (pykafka.topic.Topic method), 71
__weakref__ (pykafka.balancedconsumer.BalancedConsumer attribute), 18
__weakref__ (pykafka.broker.Broker attribute), 21
__weakref__ (pykafka.client.KafkaClient attribute), 25
__weakref__ (pykafka.cluster.Cluster attribute), 26
__weakref__ (pykafka.common.CompressionType attribute), 28
__weakref__ (pykafka.common.OffsetType attribute), 28
__weakref__ (pykafka.connection.BrokerConnection attribute), 29
__weakref__ (pykafka.connection.SslConfig attribute), 29
__weakref__ (pykafka.exceptions.KafkaException attribute), 31
__weakref__ (pykafka.exceptions.UnicodeException attribute), 33
__weakref__ (pykafka.handlers.Handler attribute), 34
__weakref__ (pykafka.handlers.RequestHandler attribute), 37
__weakref__ (pykafka.handlers.ResponseFuture attribute), 33
__weakref__ (pykafka.handlers.ThreadingHandler.Semaphore attribute), 36
__weakref__ (pykafka.partition.Partition attribute), 43
__weakref__ (pykafka.partitioners.BasePartitioner attribute), 44
__weakref__ (pykafka.producer.Producer attribute), 47
__weakref__ (pykafka.protocol.ConsumerGroupProtocolMetadata attribute), 63
__weakref__ (pykafka.protocol.MemberAssignment attribute), 63
__weakref__ (pykafka.protocol.MessageSet attribute), 60
__weakref__ (pykafka.simpleconsumer.SimpleConsumer attribute), 69
__weakref__ (pykafka.topic.Topic attribute), 71
_add_partitions() (pykafka.balancedconsumer.BalancedConsumer method), 18
_add_self() (pykafka.balancedconsumer.BalancedConsumer method), 18
_asdict() (pykafka.handlers.RequestHandler.Shared method), 36
_asdict() (pykafka.handlers.RequestHandler.Task method), 37
_asdict() (pykafka.membershipprotocol.GroupMembershipProtocol method), 42
_asdict() (pykafka.protocol.FetchPartitionResponse method), 52
_asdict() (pykafka.protocol.OffsetCommitPartitionResponse method), 54
_asdict() (pykafka.protocol.OffsetFetchPartitionResponse method), 55
_asdict() (pykafka.protocol.ProducePartitionResponse method), 64
_auto_commit() (pykafka.simpleconsumer.SimpleConsumer method), 69
_build_default_error_handlers() (pykafka.managedbalancedconsumer.ManagedBalancedConsume

method), 41
_build_default_error_handlers() (pykafka.simpleconsumer.SimpleConsumer method), 69
_build_watch_callback() (pykafka.balancedconsumer.BalancedConsumer method), 19
_discover_group_coordinator() (pykafka.simpleconsumer.SimpleConsumer method), 69
_get_broker_connection_info() (pykafka.cluster.Cluster method), 26
_get_brokers_from_zookeeper() (pykafka.cluster.Cluster method), 26
_get_compressed() (pykafka.protocol.MessageSet method), 60
_get_held_partitions() (pykafka.balancedconsumer.BalancedConsumer method), 19
_get_internal_consumer() (pykafka.balancedconsumer.BalancedConsumer method), 19
_get_metadata() (pykafka.cluster.Cluster method), 27
_get_participants() (pykafka.balancedconsumer.BalancedConsumer method), 19
_get_unique_req_handler() (pykafka.broker.Broker method), 21
_handle_error() (pykafka.managedbalancedconsumer.ManagedBalancedConsumer method), 41
_join_group() (pykafka.managedbalancedconsumer.ManagedBalancedConsumer method), 41
_legacy_wrap_socket() (pykafka.connection.SslConfig method), 29
_make() (pykafka.handlers.RequestHandler.Shared class method), 37
_make() (pykafka.handlers.RequestHandler.Task class method), 37
_make() (pykafka.membershipprotocol.GroupMembershipProtocol class method), 42
_make() (pykafka.protocol.FetchPartitionResponse class method), 52
_make() (pykafka.protocol.OffsetCommitPartitionResponse class method), 54
_make() (pykafka.protocol.OffsetFetchPartitionResponse class method), 55
_make() (pykafka.protocol.ProducePartitionResponse class method), 64
_partitions (pykafka.balancedconsumer.BalancedConsumer attribute), 19
_path_from_partition() (pykafka.balancedconsumer.BalancedConsumer method), 19
_path_self (pykafka.balancedconsumer.BalancedConsumer attribute), 19
_produce() (pykafka.producer.Producer method), 47
_produce_has_timed_out() (pykafka.producer.Producer method), 48
_raise_worker_exceptions() (pykafka.balancedconsumer.BalancedConsumer method), 19
_raise_worker_exceptions() (pykafka.producer.Producer method), 48
_raise_worker_exceptions() (pykafka.simpleconsumer.SimpleConsumer method), 69
_rebalance() (pykafka.balancedconsumer.BalancedConsumer method), 19
_remove_partitions() (pykafka.balancedconsumer.BalancedConsumer method), 19
_replace() (pykafka.handlers.RequestHandler.Shared method), 37
_replace() (pykafka.handlers.RequestHandler.Task method), 37
_replace() (pykafka.membershipprotocol.GroupMembershipProtocol method), 42
_replace() (pykafka.protocol.FetchPartitionResponse method), 52
_replace() (pykafka.protocol.OffsetCommitPartitionResponse method), 54
_replace() (pykafka.protocol.OffsetFetchPartitionResponse method), 55
_replace() (pykafka.protocol.ProducePartitionResponse method), 64
_request_random_broker() (pykafka.cluster.Cluster method), 27
_send_request() (pykafka.producer.Producer method), 48
_set_watches() (pykafka.balancedconsumer.BalancedConsumer method), 19
_setup_autocommit_worker() (pykafka.simpleconsumer.SimpleConsumer method), 69
_setup_fetch_workers() (pykafka.simpleconsumer.SimpleConsumer method), 69
_setup_heartbeat_worker() (pykafka.managedbalancedconsumer.ManagedBalancedConsumer method), 41
_setup_internal_consumer() (pykafka.balancedconsumer.BalancedConsumer method), 19
_setup_owned_brokers() (pykafka.producer.Producer method), 48
_setup_zookeeper() (pykafka.balancedconsumer.BalancedConsumer method), 19
_start_thread() (pykafka.handlers.RequestHandler method), 37
_sync_group() (pykafka.managedbalancedconsumer.ManagedBalancedConsumer method), 41
_unpack_message_set() (pykafka.protocol.FetchResponse method), 52
_update() (pykafka.producer.Producer method), 48
_update() (pykafka.simpleconsumer.SimpleConsumer

method), 69
_update_brokers() (pykafka.cluster.Cluster method), 27
_update_member_assignment() (pykafka.balancedconsumer.BalancedConsumer method), 19
_update_member_assignment() (pykafka.managedbalancedconsumer.ManagedBalancedConsumer method), 41
_wait_all() (pykafka.producer.Producer method), 48
_wait_for_slot_available() (pykafka.simpleconsumer.SimpleConsumer method), 69

A

acquire() (pykafka.handlers.ThreadingHandler.Semaphore method), 36
add_message() (pykafka.protocol.ProduceRequest method), 50
add_request() (pykafka.protocol.FetchRequest method), 51
ApiVersionsRequest (class in pykafka.protocol), 61
ApiVersionsResponse (class in pykafka.protocol), 61

B

BalancedConsumer (class in pykafka.balancedconsumer), 16
BasePartitioner (class in pykafka.partitioners), 44
Broker (class in pykafka.broker), 20
BrokerConnection (class in pykafka.connection), 29
brokers (pykafka.cluster.Cluster attribute), 27

C

Cluster (class in pykafka.cluster), 26
commit_consumer_group_offsets() (pykafka.broker.Broker method), 21
commit_offsets() (pykafka.balancedconsumer.BalancedConsumer method), 20
commit_offsets() (pykafka.simpleconsumer.SimpleConsumer method), 69
CompressionType (class in pykafka.common), 28
connect() (pykafka.broker.Broker method), 22
connect() (pykafka.connection.BrokerConnection method), 30
connect_offsets_channel() (pykafka.broker.Broker method), 22
connected (pykafka.broker.Broker attribute), 22
connected (pykafka.connection.BrokerConnection attribute), 30
connection (pykafka.handlers.RequestHandler.Shared attribute), 37
consume() (pykafka.balancedconsumer.BalancedConsumer method), 20
consume() (pykafka.simpleconsumer.SimpleConsumer method), 69
ConsumerGroupProtocolMetadata (class in pykafka.protocol), 62
ConsumerStoppedException, 30
CreateTopicRequest (class in pykafka.protocol), 63
CreateTopicsRequest (class in pykafka.protocol), 61
CreateTopicsResponse (class in pykafka.protocol), 61

D

decide_partitions (pykafka.membershipprotocol.GroupMembershipProtocol attribute), 42
decide_partitions_range() (in module pykafka.membershipprotocol), 42
decide_partitions_roundrobin() (in module pykafka.membershipprotocol), 42
decode() (pykafka.protocol.MessageSet class method), 60
decode_gzip() (in module pykafka.utils.compression), 73
decode_lz4_old_kafka() (in module pykafka.utils.compression), 73
decode_snappy() (in module pykafka.utils.compression), 73
DeleteTopicsRequest (class in pykafka.protocol), 62
DeleteTopicsResponse (class in pykafka.protocol), 62
DescribeGroupsRequest (class in pykafka.protocol), 58
DescribeGroupsResponse (class in pykafka.protocol), 58
disconnect() (pykafka.connection.BrokerConnection method), 30

E

earliest_available_offset() (pykafka.partition.Partition method), 43
earliest_available_offsets() (pykafka.topic.Topic method), 71
empty() (pykafka.handlers.ThreadingHandler.Queue method), 34
encode_gzip() (in module pykafka.utils.compression), 72
encode_lz4_old_kafka() (in module pykafka.utils.compression), 73
encode_snappy() (in module pykafka.utils.compression), 73
ending (pykafka.handlers.RequestHandler.Shared attribute), 37
err (pykafka.protocol.FetchPartitionResponse attribute), 52
err (pykafka.protocol.OffsetCommitPartitionResponse attribute), 54
err (pykafka.protocol.OffsetFetchPartitionResponse attribute), 56
err (pykafka.protocol.ProducePartitionResponse attribute), 64
Event() (pykafka.handlers.ThreadingHandler method), 34

F

fetch() (pykafka.simpleconsumer.SimpleConsumer method), 70

fetch_api_versions() (pykafka.cluster.Cluster method), 27
fetch_consumer_group_offsets() (pykafka.broker.Broker method), 22
fetch_offset_limit() (pykafka.partition.Partition method), 43
fetch_offset_limits() (pykafka.topic.Topic method), 71
fetch_offsets() (pykafka.simpleconsumer.SimpleConsumer method), 70
FetchPartitionResponse (class in pykafka.protocol), 51
FetchRequest (class in pykafka.protocol), 51
FetchResponse (class in pykafka.protocol), 52
FetchResponseV1 (class in pykafka.protocol), 63
FetchResponseV2 (class in pykafka.protocol), 63
from_metadata() (pykafka.broker.Broker class method), 22
full() (pykafka.handlers.ThreadingHandler.Queue method), 34
future (pykafka.handlers.RequestHandler.Task attribute), 37

G

GaiError (pykafka.handlers.ThreadingHandler attribute), 34
get() (pykafka.handlers.ResponseFuture method), 33
get() (pykafka.handlers.ThreadingHandler.Queue method), 34
get_balanced_consumer() (pykafka.topic.Topic method), 72
get_bytes() (pykafka.protocol.ApiVersionsRequest method), 61
get_bytes() (pykafka.protocol.CreateTopicsRequest method), 61
get_bytes() (pykafka.protocol.DeleteTopicsRequest method), 62
get_bytes() (pykafka.protocol.DescribeGroupsRequest method), 58
get_bytes() (pykafka.protocol.FetchRequest method), 51
get_bytes() (pykafka.protocol.GroupCoordinatorRequest method), 53
get_bytes() (pykafka.protocol.HeartbeatRequest method), 57
get_bytes() (pykafka.protocol.JoinGroupRequest method), 56
get_bytes() (pykafka.protocol.LeaveGroupRequest method), 57
get_bytes() (pykafka.protocol.ListGroupsRequest method), 58
get_bytes() (pykafka.protocol.ListOffsetRequest method), 53
get_bytes() (pykafka.protocol.ListOffsetRequestV1 method), 64
get_bytes() (pykafka.protocol.MetadataRequest method), 49
get_bytes() (pykafka.protocol.MetadataRequestV4 method), 66
get_bytes() (pykafka.protocol.OffsetCommitRequest method), 54
get_bytes() (pykafka.protocol.OffsetFetchRequest method), 55
get_bytes() (pykafka.protocol.ProduceRequest method), 50
get_bytes() (pykafka.protocol.SyncGroupRequest method), 57
get_delivery_report() (pykafka.producer.Producer method), 48
get_group_coordinator() (pykafka.cluster.Cluster method), 27
get_managed_group_descriptions() (pykafka.cluster.Cluster method), 27
get_nowait() (pykafka.handlers.ThreadingHandler.Queue method), 34
get_producer() (pykafka.topic.Topic method), 72
get_simple_consumer() (pykafka.topic.Topic method), 72
get_sync_producer() (pykafka.topic.Topic method), 72
GroupAuthorizationFailed, 30
GroupCoordinatorNotAvailable, 30
GroupCoordinatorRequest (class in pykafka.protocol), 53
GroupCoordinatorResponse (class in pykafka.protocol), 53
GroupHashingPartitioner (class in pykafka.partitioners), 45
GroupLoadInProgress, 30
GroupMembershipProtocol (class in pykafka.membershipprotocol), 41

H

handle_partition_responses() (in module pykafka.utils.error_handlers), 73
Handler (class in pykafka.handlers), 34
handler (pykafka.broker.Broker attribute), 23
handler (pykafka.cluster.Cluster attribute), 27
HashingPartitioner (class in pykafka.partitioners), 44
heartbeat() (pykafka.broker.Broker method), 23
HeartbeatRequest (class in pykafka.protocol), 57
HeartbeatResponse (class in pykafka.protocol), 57
held_offsets (pykafka.balancedconsumer.BalancedConsumer attribute), 20
held_offsets (pykafka.simpleconsumer.SimpleConsumer attribute), 70
host (pykafka.broker.Broker attribute), 23

I

id (pykafka.broker.Broker attribute), 23
id (pykafka.partition.Partition attribute), 43
IllegalGeneration, 30
InconsistentGroupProtocol, 30
InvalidMessageError, 30

InvalidMessageSize, 31
InvalidSessionTimeout, 31
InvalidTopic, 31
isr (pykafka.partition.Partition attribute), 43

J

join() (pykafka.handlers.ThreadingHandler.Queue method), 34
join_group() (pykafka.broker.Broker method), 23
JoinGroupRequest (class in pykafka.protocol), 56
JoinGroupResponse (class in pykafka.protocol), 56

K

KafkaClient (class in pykafka.client), 24
KafkaException, 31

L

latest_available_offset() (pykafka.partition.Partition method), 43
latest_available_offsets() (pykafka.topic.Topic method), 72
leader (pykafka.partition.Partition attribute), 44
LeaderNotAvailable, 31
LeaderNotFoundError, 31
leave_group() (pykafka.broker.Broker method), 23
LeaveGroupRequest (class in pykafka.protocol), 57
LeaveGroupResponse (class in pykafka.protocol), 58
ListGroupsRequest (class in pykafka.protocol), 58
ListGroupsResponse (class in pykafka.protocol), 58
ListOffsetRequest (class in pykafka.protocol), 52
ListOffsetRequestV1 (class in pykafka.protocol), 64
ListOffsetResponse (class in pykafka.protocol), 53
ListOffsetResponseV1 (class in pykafka.protocol), 64
Lock() (pykafka.handlers.ThreadingHandler method), 34

M

ManagedBalancedConsumer (class in pykafka.managedbalancedconsumer), 38
max_offset (pykafka.protocol.FetchPartitionResponse attribute), 52
MemberAssignment (class in pykafka.protocol), 63
Message (class in pykafka.common), 28
Message (class in pykafka.protocol), 59
message_count() (pykafka.protocol.ProduceRequest method), 50
messages (pykafka.protocol.FetchPartitionResponse attribute), 52
messages (pykafka.protocol.ProduceRequest attribute), 50
MessageSet (class in pykafka.protocol), 60
MessageSetDecodeFailure, 31
MessageSizeTooLarge, 31
metadata (pykafka.membershipprotocol.GroupMembershipProtocol attribute), 42
metadata (pykafka.protocol.OffsetFetchPartitionResponse attribute), 56
MetadataRequest (class in pykafka.protocol), 49
MetadataRequestV1 (class in pykafka.protocol), 63
MetadataRequestV2 (class in pykafka.protocol), 65
MetadataRequestV3 (class in pykafka.protocol), 65
MetadataRequestV4 (class in pykafka.protocol), 66
MetadataRequestV5 (class in pykafka.protocol), 66
MetadataResponse (class in pykafka.protocol), 49
MetadataResponseV1 (class in pykafka.protocol), 63
MetadataResponseV2 (class in pykafka.protocol), 65
MetadataResponseV3 (class in pykafka.protocol), 65
MetadataResponseV4 (class in pykafka.protocol), 66
MetadataResponseV5 (class in pykafka.protocol), 66

N

name (pykafka.topic.Topic attribute), 72
NoBrokersAvailableError, 31
NoMessagesConsumedError, 31
NotCoordinatorForGroup, 31
NotLeaderForPartition, 31

O

offset (pykafka.protocol.OffsetFetchPartitionResponse attribute), 56
offset (pykafka.protocol.ProducePartitionResponse attribute), 64
OffsetCommitPartitionResponse (class in pykafka.protocol), 54
OffsetCommitRequest (class in pykafka.protocol), 54
OffsetCommitResponse (class in pykafka.protocol), 54
OffsetFetchPartitionResponse (class in pykafka.protocol), 55
OffsetFetchRequest (class in pykafka.protocol), 55
OffsetFetchRequestV1 (class in pykafka.protocol), 64
OffsetFetchRequestV2 (class in pykafka.protocol), 65
OffsetFetchResponse (class in pykafka.protocol), 56
OffsetFetchResponseV1 (class in pykafka.protocol), 64
OffsetFetchResponseV2 (class in pykafka.protocol), 65
OffsetMetadataTooLarge, 32
OffsetOutOfRangeError, 32
OffsetRequestFailedError, 32
offsets_channel_connected (pykafka.broker.Broker attribute), 24
offsets_channel_handler (pykafka.broker.Broker attribute), 24
OffsetType (class in pykafka.common), 28

P

pack_into() (pykafka.protocol.Message method), 59
pack_into() (pykafka.protocol.MessageSet method), 60
Partition (class in pykafka.partition), 43
PartitionFetchRequest (class in pykafka.protocol), 50

PartitionOffsetCommitRequest (class in pykafka.protocol), 54
PartitionOffsetFetchRequest (class in pykafka.protocol), 55
PartitionOffsetRequest (class in pykafka.protocol), 62
PartitionOwnedError, 32
partitions (pykafka.balancedconsumer.BalancedConsumer attribute), 20
partitions (pykafka.simpleconsumer.SimpleConsumer attribute), 70
partitions (pykafka.topic.Topic attribute), 72
port (pykafka.broker.Broker attribute), 24
produce() (pykafka.producer.Producer method), 48
ProduceFailureError, 32
ProducePartitionResponse (class in pykafka.protocol), 63
Producer (class in pykafka.producer), 45
ProduceRequest (class in pykafka.protocol), 49
ProduceResponse (class in pykafka.protocol), 50
ProducerQueueFullError, 32
ProducerStoppedException, 32
protocol_name (pykafka.membershipprotocol.GroupMembershipProtocol attribute), 42
protocol_type (pykafka.membershipprotocol.GroupMembershipProtocol attribute), 42
ProtocolClientError, 32
put() (pykafka.handlers.ThreadingHandler.Queue method), 35
put_nowait() (pykafka.handlers.ThreadingHandler.Queue method), 35
pykafka.balancedconsumer (module), 16
pykafka.broker (module), 20
pykafka.client (module), 24
pykafka.cluster (module), 26
pykafka.common (module), 28
pykafka.connection (module), 28
pykafka.exceptions (module), 30
pykafka.handlers (module), 33
pykafka.managedbalancedconsumer (module), 38
pykafka.membershipprotocol (module), 41
pykafka.partition (module), 43
pykafka.partitioners (module), 44
pykafka.producer (module), 45
pykafka.protocol (module), 49
pykafka.simpleconsumer (module), 67
pykafka.topic (module), 70
pykafka.utils.compression (module), 72
pykafka.utils.error_handlers (module), 73
pykafka.utils.socket (module), 74
pykafka.utils.struct_helpers (module), 74

Q

qsize() (pykafka.handlers.ThreadingHandler.Queue method), 35

R

raise_error() (in module pykafka.utils.error_handlers), 74
RandomPartitioner (class in pykafka.partitioners), 44
RdKafkaException, 32
RdKafkaStoppedException, 32
RebalanceInProgress, 32
reconnect() (pykafka.connection.BrokerConnection method), 30
recvall_into() (in module pykafka.utils.socket), 74
release() (pykafka.handlers.ThreadingHandler.Semaphore method), 36
replicas (pykafka.partition.Partition attribute), 44
request (pykafka.handlers.RequestHandler.Task attribute), 37
request() (pykafka.connection.BrokerConnection method), 30
request() (pykafka.handlers.RequestHandler method), 37
RequestHandler (class in pykafka.handlers), 36
RequestHandler.Shared (class in pykafka.handlers), 36
RequestHandler.Task (class in pykafka.handlers), 37
requests (pykafka.handlers.RequestHandler.Shared attribute), 37
RequestTimedOut, 33
reset_offsets() (pykafka.balancedconsumer.BalancedConsumer method), 20
reset_offsets() (pykafka.simpleconsumer.SimpleConsumer method), 70
response() (pykafka.connection.BrokerConnection method), 30
ResponseFuture (class in pykafka.handlers), 33

S

set_error() (pykafka.handlers.ResponseFuture method), 34
set_response() (pykafka.handlers.ResponseFuture method), 34
SimpleConsumer (class in pykafka.simpleconsumer), 67
SockErr (pykafka.handlers.ThreadingHandler attribute), 36
Socket (pykafka.handlers.ThreadingHandler attribute), 36
SocketDisconnectedError, 33
spawn() (pykafka.handlers.Handler method), 34
spawn() (pykafka.handlers.ThreadingHandler method), 36
SslConfig (class in pykafka.connection), 28
start() (pykafka.balancedconsumer.BalancedConsumer method), 20
start() (pykafka.handlers.RequestHandler method), 38
start() (pykafka.managedbalancedconsumer.ManagedBalancedConsumer method), 41
start() (pykafka.producer.Producer method), 49
start() (pykafka.simpleconsumer.SimpleConsumer method), 70

stop() (pykafka.balancedconsumer.BalancedConsumer method), 20
stop() (pykafka.handlers.RequestHandler method), 38
stop() (pykafka.managedbalancedconsumer.ManagedBalancedConsumer method), 41
stop() (pykafka.producer.Producer method), 49
stop() (pykafka.simpleconsumer.SimpleConsumer method), 70
sync_group() (pykafka.broker.Broker method), 24
SyncGroupRequest (class in pykafka.protocol), 56
SyncGroupResponse (class in pykafka.protocol), 57

T

task_done() (pykafka.handlers.ThreadingHandler.Queue method), 35
ThreadingHandler (class in pykafka.handlers), 34
ThreadingHandler.Queue (class in pykafka.handlers), 34
ThreadingHandler.Semaphore (class in pykafka.handlers), 35
timestamp_dt (pykafka.protocol.Message attribute), 60
Topic (class in pykafka.topic), 70
topic (pykafka.balancedconsumer.BalancedConsumer attribute), 20
topic (pykafka.partition.Partition attribute), 44
topic (pykafka.simpleconsumer.SimpleConsumer attribute), 70
TopicAuthorizationFailed, 33
topics (pykafka.cluster.Cluster attribute), 27

U

UnicodeException, 33
UnknownError, 33
UnknownMemberId, 33
UnknownTopicOrPartition, 33
unpack_from() (in module pykafka.utils.struct_helpers), 74
update() (pykafka.cluster.Cluster method), 27
update() (pykafka.partition.Partition method), 44
update() (pykafka.topic.Topic method), 72
update_cluster() (pykafka.client.KafkaClient method), 25

W

wrap_socket() (pykafka.connection.SslConfig method), 29