

---

# **pykafka**

***Release 2.6.1.dev1***

**Aug 10, 2017**



---

## Contents

---

<b>1</b>	<b>Getting Started</b>	<b>3</b>
<b>2</b>	<b>Using the librdkafka extension</b>	<b>5</b>
<b>3</b>	<b>Operational Tools</b>	<b>7</b>
<b>4</b>	<b>What happened to Samsa?</b>	<b>9</b>
<b>5</b>	<b>PyKafka or kafka-python?</b>	<b>11</b>
<b>6</b>	<b>Contributing</b>	<b>13</b>
<b>7</b>	<b>Support</b>	<b>15</b>
	<b>Python Module Index</b>	<b>65</b>



PyKafka is a cluster-aware Kafka $\geq$ 0.8.2 client for Python. It includes Python implementations of Kafka producers and consumers, which are optionally backed by a C extension built on [librdkafka](#), and runs under Python 2.7+, Python 3.4+, and PyPy.

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
```

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
```



---

## 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
{'my.test': <pykafka.topic.Topic at 0x19bc8c0 (name=my.test)>}
>>> 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. 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.

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



---

### 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.

We currently test against librdkafka 0.9.1 only. Note that use on pypy is not recommended at this time; the producer is certainly expected to crash.



## 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 found by running

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

or after installing PyKafka via `setuptools` or `pip`:

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



---

### What happened to Samsa?

---

This project used to be called samsa. It has been renamed PyKafka and has been fully overhauled to support Kafka 0.8.2. We chose to target 0.8.2 because the offset Commit/Fetch API stabilized on that release.

The Samsa [PyPI package](#) will stay up for the foreseeable future and tags for previous versions will always be available in this repo.



## CHAPTER 5

---

### PyKafka or kafka-python?

---

These are two different projects. See [the discussion here](#) for comparisons between the two projects.





## CHAPTER 6

---

### 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](#).



If you need help using PyKafka or have found a bug, please open a [github issue](#) or use the [Google Group](#).

## Help Documents

### PyKafka Usage Guide

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

### Consumer Patterns

#### Setting the initial offset

This section applies to both the *SimpleConsumer* and the *BalancedConsumer*.

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](#).

## Producer Patterns

TODO

## Kafka 0.9 Roadmap for PyKafka

Date: November 20, 2015

### Quick summary

The current stable version of Kafka is 0.8.2. This is meant to run against the latest Zookeeper versions, e.g. 3.4.6.

The latest releases of pykafka target 0.8.2 **specifically**; the Python code is not backwards compatible with 0.8.1 due to changes in what is known as Offset Commit/Fetch API, which pykafka uses to simplify the offset management APIs and standardize them with other clients that talk to Kafka.

The 0.8.2 release will likely be the most stable Kafka broker to use in production for the next couple of months. However, as we will discuss later, there is a specific bug in Kafka brokers that was fixed in 0.9.0 that we may find advantageous to backport to 0.8.2.

Meanwhile, 0.9.0 is “around the corner” (currently in release candidate form) and introduces, yet again, a brand new consumer API, which we need to track and wrap in pykafka. But for that to stabilize will take some time.

## SimpleConsumer vs BalancedConsumer

Why does pykafka exist? That's a question I sometimes hear from people, especially since there are alternative implementations of the Kafka protocol floating around in the Python community, notably [kafka-python](#).

One part of the reason pykafka exists is to build a more Pythonic API for working with Kafka that supports every major Python interpreter (Python 2/3, PyPy) and every single Kafka feature. We also have an interest in making Kafka consumers fast, with C optimizations for protocol speedups. But the **real** reason it exists is to implement a **scalable and reliable BalancedConsumer** implementation atop Kafka and Zookeeper. This was missing from any Kafka and Python project, and we (and many other users) desperately needed it to use Kafka in the way it is meant to be used.

Since there is some confusion on this, let's do a crystal clear discussion of the differences between these two consumer types.

**SimpleConsumer** communicates **directly** with a Kafka broker to consume a Kafka topic, and takes "ownership" of 100% of the partitions reported for that topic. It does round-robin consumption of messages from those partitions, while using the aforementioned Commit/Fetch API to manage offsets. Under the hood, the Kafka broker talks to Zookeeper to maintain the offset state.

The main problems with SimpleConsumer: scalability, parallelism, and high availability. If you have a busy topic with lots of partitions, a SimpleConsumer may not be able to keep up, and your offset lag (as reported by kafka-tools) will constantly be behind, or worse, may grow over time. You may also have code that needs to react to messages, and that code may be CPU-bound, so you may be seeking to achieve multi-core or multi-node parallelism. Since a SimpleConsumer has no coordination mechanism, you have no options here: multiple SimpleConsumer instances reading from the same topic will read **the same messages** from that topic – that is, the data won't be spread evenly among the consumers. Finally, there is the availability concern. If your SimpleConsumer dies, your pipeline dies. You'd ideally like to have several consumers such that the death of one does not result in the death of your pipeline.

One other side note related to using Kafka in Storm, since that's a common use case. Typically Kafka data enters a Storm topology via a Spout written against pykafka's API. If that Spout makes use of a SimpleConsumer, you can only set that Spout's parallelism level to 1 – a parallel Spout will emit duplicate tuples into your topology!

So, now let's discuss **BalancedConsumer** and how it solves these problems. Instead of taking ownership of 100% partitions upon consumption of a topic, a BalancedConsumer in Kafka 0.8.2 coordinates the state for several consumers who "share" a single topic by talking to the Kafka broker and directly to Zookeeper. It figures this out by registering a "consumer group ID", which is an identifier associated with several consumer processes that are all eating data from the same topic, in a balanced manner.

The following discussion of the BalancedConsumer operation is very simplified and high-level – it's not exactly how it works. But it'll serve to illustrate the idea. Let's say you have 10 partitions for a given topic. A BalancedConsumer connects asks the cluster, "what partitions are available?". The cluster replies, "10". So now that consumer takes "ownership" of 100% of the partitions, and starts consuming. At this moment, the BalancedConsumer is operating similarly to a SimpleConsumer.

Then a **second** BalancedConsumer connects and the cluster, "which partitions are available? Cluster replies, "0", and asks the BalancedConsumer to wait a second. It now initiates a "partition rebalancing". This is a fancy dance between Zookeeper and Kafka, but the end result is that 5 partitions get "owned" by consumer A and 5 get "owned" by consumer B. The original consumer receives a notification that the partition balancing has changed, so it now consumes from fewer partitions. Meanwhile, the second BalancedConsumer now gets a new notification: "5" is the number of partitions it can now own. At this point, 50% of the stream is being consumed by consumer A, and 50% by consumer B.

You can see where this goes. A third, fourth, fifth, or sixth BalancedConsumer could join the group. This would split up the partitions yet further. However, note – we mentioned that the total number of partitions for this topic was 10. Thus, though balancing will work, it will only work up to the number of total partitions available for a topic. That is, if we had 11 BalancedConsumers in this consumer group, we'd have one idle consumer and 10 active consumers, with the active ones only consuming 1 partition each.

The good news is, it's very typical to run Kafka topics with 20, 50, or even 100 partitions per topic, and this typically provides enough parallelism and availability for almost any need.

Finally, availability is provided with the same mechanism. If you unplug a `BalancedConsumer`, its partitions are returned to the group, and other group members can take ownership. This is especially powerful in a Storm topology, where a Spout using a `BalancedConsumer` might have parallelism of 10 or 20, and single Spout instance failures would trigger rebalancing automatically.

## Pure Python vs rdkafka

A commonly used Kafka utility is `kafkacat`, which is written by Magnus Edenhill. It is written in C and makes use of the `librdkafka` library, which is a pure C wrapper for the Kafka protocol that has been benchmarked to support 3 million messages per second on the consumer side. A member of the Parse.ly team has written a `pykafka` binding for this library which serves two purposes: a) speeding up Python consumers and b) providing an alternative protocol implementation that allows us to isolate protocol-level bugs.

Note that on the consumer side, `librdkafka` only handles direct communication with the Kafka broker. Therefore, `BalancedConsumer` still makes use of `pykafka`'s pure Python Zookeeper handling code to implement partition rebalancing among consumers.

Under the hood, `librdkafka` is wrapped using Python's C extension API, therefore it adds a little C wrapper code to `pykafka`'s codebase. Building this C extension requires that `librdkafka` is already built and installed on your machine (local or production).

By the end of November, `rdkafka` will be a fully supported option of `pykafka`. This means `SimpleConsumers` can be sped up to handle larger streams without rebalancing, and it also means `BalancedConsumer`'s get better per-core or per-process utilization. Making use of this protocol is as simple as passing a `use_rdkafka=True` flag to the appropriate consumer or producer creation functions.

## Compatibility Matrix

Kafka lacks a coherent release management process, which is one of the worst parts of the project. Minor dot-version releases have dramatically changed client protocols, thus resembling major version changes to client teams working on projects like `pykafka`. To help sort through the noise, here is a compatibility matrix for Kafka versions of whether we have protocol support for these versions in latest stable versions of our consumer/producer classes:

Kafka version	pykafka?	rdkafka?
0.8.1	No	No
0.8.2	Yes	Yes
0.9.0	Planned	Planned

Note that 0.9.0.0 is currently in "release candidate" stage as of November 2015.

## Core Kafka Issues On Our Radar

There are several important Kafka core issues that are on our radar and that have changed things dramatically (hopefully for the better) in the new Kafka 0.9.0 release version. These are summarized in this table:

Issue	0.8.2	0.9.0	Link?
New Consumer API	N/A	Added	<a href="#">KAFKA-1328</a>
New Consumer API Extras	N/A	In Flux	<a href="#">KAFKA-1326</a>
Security/SSL	N/A	Added	<a href="#">KAFKA-1682</a>
Broker/ZK Crash	Bug	Fixed	<a href="#">KAFKA-1387</a>
Documentation	"Minimal"	"Improved"	<a href="#">New Docs</a>

Let's focus on three areas here: new consumer API, security, and broker/ZK crash.

## New Consumer API

One of the biggest new features of Kafka 0.9.0 is a brand new Consumer API. The good news **may** be that despite introducing this new API, they **may** still support their “old” APIs that were stabilized in Kafka 0.8.2. We are going to explore this as this would provide a smoother upgrade path for pykafka users for certain.

The main difference for this new API is moving more of the `BalancedConsumer` partition rebalancing logic into the broker itself. This would certainly be a good idea to standardize how `BalancedConsumers` work across programming languages, but we don’t have a lot of confidence that this protocol is bug-free at the moment. The Kafka team even describes **their own** 0.9.0 consumer as being “beta quality”.

## Security/SSL

This is one of Kafka’s top requests. To provide secure access to Kafka topics, people have had to use the typical IP whitelisting and VPN hacks, which is problematic since they can often impact the overall security of a system, impact performance, and are operationally complex to maintain.

The Kafka 0.9.0 release includes a standard mechanism for doing SSL-based security in communicating with Kafka brokers. We’ll need to explore what the requirements and limitations are of this scheme to see if it can be supported directly by pykafka.

## Broker/ZK Crash

This is perhaps the most annoying issue regarding this new release. We have several reports from the community of Kafka brokers that crash as a result of a coordination issue with Zookeeper. A bug fix was worked on for several months and a patched build of 0.8.1 fixed the issue permanently for some users, but because the Kafka community cancelled a 0.8.3 release, favoring 0.9.0 instead, no patched build of 0.8.2 was ever created. This issue **is** fixed in 0.9.0, however.

## The Way Forward

We want pykafka to support 0.8.2 and 0.9.0 in a single source tree. We’d like the `rdkafka` implementation to have similar support. We think this will likely be supported **without** using Kafka’s 0.9.0 “New Consumer API”. This will give users a 0.9.0 upgrade path for stability (fixing the Broker/ZK Crash, and allowing use of `SimpleConsumer`, `BalancedConsumer`, and C-optimized versions with `rdkafka`).

We don’t know, yet, whether the new Security/SSL scheme requires use of the new Consumer APIs. If so, the latter may be a blocker for the former. We will likely discover the answer to this in November 2015.

A [tracker issue for Kafka 0.9.0 support](#) in pykafka was opened, and that’s where discussion should go for now.

## 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.*

## 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='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)
```

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='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)
```

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** (*bytes*) – The name of the consumer group this consumer should join.
- **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.
- **zookeeper\_connection\_timeout\_ms** (*int*) – The maximum time (in milliseconds) that the consumer waits while establishing a connection to zookeeper.
- **zookeeper\_connect** (*str*) – Comma-separated (ip1:port1,ip2:port2) strings indicating the zookeeper nodes 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-ids, 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.

**`__iter__`** ()

Yield an infinite stream of messages until the consumer times out

**`__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>

**`_decide_partitions`** (*participants, consumer\_id=None*)

Decide which partitions belong to this consumer.

Uses the consumer rebalancing algorithm described here [https://kafka.apache.org/documentation/#impl\\_consumerrebalance](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 other consumers in this consumer group.
- **`consumer_id`** – The ID of the consumer for which to generate a partition assignment. Defaults to `self._consumer_id`

**`_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`** ()

Commit offsets for this consumer’s partitions

Uses the offset commit/fetch API

**`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

Issue an OffsetRequest for each partition and set the appropriate returned offset in the consumer's internal offset counter.

**Parameters** `partition_offsets` (Sequence of tuples of the form (`pykafka.partition.Partition`, int)) – (`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 of the message whose offset the partition should have OR the new offset the partition should have

NOTE: If an instance of `timestamp_or_offset` is treated by kafka as an invalid offset timestamp, this function directly sets the consumer's internal offset counter for that partition to that instance of `timestamp_or_offset`. On the next fetch request, the consumer attempts to fetch messages starting from that offset. See the following link for more information on what kafka treats as a valid offset timestamp: <https://cwiki.apache.org/confluence/display/KAFKA/A+Guide+To+The+Kafka+Protocol#AGuideToTheKafkaProtocol-OffsetRequest>

**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

## pykafka.broker

Author: Keith Bourgoïn, Emmett Butler

```
class pykafka.broker.Broker(id_, host, port, handler, socket_timeout_ms, off-
                           sets_channel_socket_timeout_ms, buffer_size=1048576,
                           source_host='', source_port=0, ssl_config=None, bro-
                           ker_version='0.9.0')
```

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, bro-
         ker_version='0.9.0')
```

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

**\_\_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** ()

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** ()

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'*)

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

#### 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*)

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 meta-data

**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.

## pykafka.client

Author: Keith Bourgoïn, 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](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*) – ZooKeeper-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.

**\_\_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.

## pykafka.cluster

```
class pykafka.cluster.Cluster(hosts, handler, socket_timeout_ms=30000, off-
                              sets_channel_socket_timeout_ms=10000, ex-
                              clude_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, ex-
         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.

**\_\_weakref\_\_**

list of weak references to the object (if defined)

**`_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_metadata`** (*broker\_connects, topics*)

Request broker metadata from a set of brokers

Returns the result of the first successful metadata 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

**`_update_brokers`** (*broker\_metadata*)

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.

#### **brokers**

The dict of known brokers for this cluster

**`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

**`update`** ()

Update known brokers and topics.

## **pykafka.common**

Author: Keith Bourgoïn

**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)

## 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*)

Connect to the broker.

**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

## pykafka.exceptions

Author: Keith Bourgoïn, 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.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.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.

**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.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.

**exception** `pykafka.exceptions.UnknownTopicOrPartition`

Bases: `pykafka.exceptions.ProtocolClientError`

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

## pykafka.handlers

Author: Keith Bourgoïn, 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*)

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** (*\*args, \*\*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.

**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  $\leq 0$ , 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** ThreadingHandler.**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.

#### `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.

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

```
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 (_self, **kwargs)
    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 RequestHandler.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 (_self, **kwargs)
        Return a new Task object replacing specified fields with new values

    future
        Alias for field number 1

    request
        Alias for field number 0

RequestHandler.__init__ (handler, connection)

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

RequestHandler.__start_thread ()
    Run the request processor

```

RequestHandler.**request** (*request*, *has\_response=True*)

Construct a new request

**Parameters** *has\_response* – Whether this request will return a response

**Returns** *pykafka.handlers.ResponseFuture*

RequestHandler.**start** ()

Start the request processor.

RequestHandler.**stop** ()

Stop the request processor.

## 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,
                                                                off-sets_channel_backoff_ms=1000,
                                                                off-sets_commit_max_retries=5,
                                                                auto_offset_reset=-2,
                                                                consumer_timeout_ms=-1,
                                                                rebal-ance_max_retries=5,
                                                                rebal-ance_backoff_ms=2000,
                                                                auto_start=True, re-set_offset_on_start=False,
                                                                post_rebalance_callback=None,
                                                                use_rdkafka=False,
                                                                com-pacted_topic=True,
                                                                heart-beat_interval_ms=3000)
```

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, off-
         sets_commit_max_retries=5, auto_offset_reset=-2, consumer_timeout_ms=-1,
         rebalance_max_retries=5, rebalance_backoff_ms=2000, auto_start=True, re-
         set_offset_on_start=False, post_rebalance_callback=None, use_rdkafka=False, com-
         pacted_topic=True, heartbeat_interval_ms=3000)

```

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** (`bytes`) – The name of the consumer group this consumer should join.
- **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

**`__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

## pykafka.partition

Author: Keith Bourgoïn, 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.

**\_\_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

**\_\_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** (*int*) – Return an offset from before this timestamp (in milliseconds)
- **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

## pykafka.partitioners

Author: Keith Bourgoïn, Emmett Butler

`pykafka.partitioners.random_partitioner` (*partitions, key*)

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

**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.

**\_\_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`.

## pykafka.producer

**class** `pykafka.producer.Producer` (*cluster, topic, partitioner=<function random\_partitioner>, 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, block\_on\_queue\_full=True, max\_request\_size=1000012, sync=False, delivery\_reports=False, auto\_start=True*)

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=<function random_partitioner>, 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,
         block_on_queue_full=True, max_request_size=1000012, sync=False, deliv-
         ery_reports=False, auto_start=True)
```

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.
- **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 parameter 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.
- **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*.
- **auto\_start** (*bool*) – Whether the producer should begin communicating with kafka after *\_\_init\_\_* is complete. If false, communication can be started with *start()*.

#### **`__weakref__`**

list of weak references to the object (if defined)

#### **`__produce`** (*message*)

Enqueue a message for the relevant broker

**Parameters** **message** (*pykafka.protocol.Message*) – Message with valid *partition\_id*, ready to be sent

#### **`__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

“Pending” messages are those that have been used in calls to *produce* and have not yet been dequeued and sent to 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 dequeuing 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

**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

## pykafka.protocol

**class** `pykafka.protocol.MetadataRequest` (*topics=None*)

Bases: `pykafka.protocol.Request`

Metadata Request

Specification:

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

**API\_KEY**

API\_KEY for this request, from the Kafka docs

**\_\_init\_\_** (*topics=None*)

Create a new MetadataRequest

**Parameters** **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

**Return type** bytearray

**class** `pykafka.protocol.MetadataResponse` (*buff*)

Bases: `pykafka.protocol.Response`

Response from MetadataRequest

Specification:

```

MetadataResponse => [Broker][TopicMetadata]
  Broker => NodeId Host Port
  NodeId => int32
  Host => string
  Port => int32
  TopicMetadata => TopicErrorCode TopicName [PartitionMetadata]
  TopicErrorCode => int16
  PartitionMetadata => PartitionErrorCode PartitionId Leader Replicas Isr
  PartitionErrorCode => int16
  PartitionId => int32
  Leader => int32
  Replicas => [int32]
  Isr => [int32]

```

**\_\_init\_\_** (*buff*)

Deserialize into a new Response

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

**class** pykafka.protocol.**ProduceRequest** (*compression\_type=0, required\_acks=1, timeout=10000*)

Bases: [pykafka.protocol.Request](#)

Produce Request

Specification:

```

ProduceRequest => RequiredAcks Timeout [TopicName [Partition MessageSetSize,
↔MessageSet]]
  RequiredAcks => int16
  Timeout => int32
  Partition => int32
  MessageSetSize => int32

```

#### API\_KEY

API\_KEY for this request, from the Kafka docs

**\_\_init\_\_** (*compression\_type=0, required\_acks=1, timeout=10000*)

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

**Parameters**

- **messages** – an iterable of `kafka.common.Message` to add
- **topic\_name** – the name of the topic to publish to
- **partition\_id** – the partition to publish to

**get\_bytes** ()  
 Serialize the message

**Returns** Serialized message

**Return type** `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.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

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

**class** `pykafka.protocol.OffsetRequest` (*partition\_requests*)  
 Bases: `pykafka.protocol.Request`

An offset request

Specification:

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

**API\_KEY**

API\_KEY for this request, from the Kafka docs

**\_\_init\_\_** (*partition\_requests*)  
 Create a new offset request

**\_\_len\_\_** ()  
 Length of the serialized message, in bytes

**get\_bytes** ()

Serialize the message

**Returns** Serialized message

**Return type** bytearray

**class** pykafka.protocol.**OffsetResponse** (*buff*)

Bases: *pykafka.protocol.Response*

An offset response

Specification:

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

**\_\_init\_\_** (*buff*)

Deserialize into a new Response

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

**class** pykafka.protocol.**OffsetCommitRequest** (*consumer\_group*, *consumer\_group\_generation\_id*, *consumer\_id*, *partition\_requests=[]*)

Bases: *pykafka.protocol.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
```

**API\_KEY**

API\_KEY for this request, from the Kafka docs

**\_\_init\_\_** (*consumer\_group*, *consumer\_group\_generation\_id*, *consumer\_id*, *partition\_requests=[]*)

Create a new offset commit request

**Parameters** **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

**Return type** bytearray

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

Bases: `pykafka.protocol.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
```

#### API\_KEY

API\_KEY for this request, from the Kafka docs

```
__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.FetchResponse (buff, offset=0)
```

Bases: `pykafka.protocol.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*)  
 Deserialize into a new Response

#### Parameters

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

**\_unpack\_message\_set** (*buff*, *partition\_id=-1*)  
 MessageSets can be nested. Get just the Messages out of it.

**static get\_subclass** (*broker\_protocol*)  
 Choose which subclass of response to demand and expect. Cf. <https://cwiki.apache.org/confluence/display/KAFKA/A+Guide+To+The+Kafka+Protocol>

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

An offset commit response

Specification:

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

**\_\_init\_\_** (*buff*)  
 Deserialize into a new Response

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

**class** `pykafka.protocol.OffsetFetchRequest` (*consumer\_group*, *partition\_requests=[]*)  
 Bases: `pykafka.protocol.Request`

An offset fetch request

Specification:

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

**API\_KEY**

API\_KEY for this request, from the Kafka docs

`__init__ (consumer_group, partition_requests=[])`

Create a new offset fetch request

**Parameters** `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

**Return type** bytearray

`class pykafka.protocol.OffsetFetchResponse (buff)`

Bases: `pykafka.protocol.Response`

An offset fetch response

Specification:

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

`__init__ (buff)`

Deserialize into a new Response

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

`class pykafka.protocol.PartitionOffsetRequest`

Bases: `pykafka.protocol.PartitionOffsetRequest`

Offset request for a specific topic/partition

**Variables**

- **topic\_name** – Name of the topic to look up
- **partition\_id** – Id of the partition to look up
- **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
- **max\_offsets** – How many offsets to return

`class pykafka.protocol.GroupCoordinatorRequest (consumer_group)`

Bases: `pykafka.protocol.Request`

A consumer metadata request

Specification:

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

#### API\_KEY

API\_KEY for this request, from the Kafka docs

`__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

**Return type** bytearray

**class** `pykafka.protocol.GroupCoordinatorResponse(buff)`

Bases: `pykafka.protocol.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

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

**class** `pykafka.protocol.PartitionOffsetCommitRequest`

Bases: `pykafka.protocol.PartitionOffsetCommitRequest`

Offset commit request for a specific topic/partition

#### Variables

- **topic\_name** – Name of the topic to look up
- **partition\_id** – Id of the partition to look up
- **offset** –
- **timestamp** –
- **metadata** – arbitrary metadata that should be committed with this offset commit

**class** `pykafka.protocol.PartitionOffsetFetchRequest`

Bases: `pykafka.protocol.PartitionOffsetFetchRequest`

Offset fetch request for a specific topic/partition

#### Variables

- **topic\_name** – Name of the topic to look up

- **partition\_id** – Id of the partition to look up

**class** `pykafka.protocol.Request`

Bases: `pykafka.utils.Serializable`

Base class for all Requests. Handles writing header information

**API\_KEY** ()

API key for this request, from the Kafka docs

**\_\_weakref\_\_**

list of weak references to the object (if defined)

**\_\_write\_header** (*buff, api\_version=0, correlation\_id=0*)

Write the header for an outgoing message.

**Parameters**

- **buff** (*buffer*) – The buffer into which to write the header
- **api\_version** (*int*) – The “kafka api version id”, used for feature flagging
- **correlation\_id** (*int*) – This is a user-supplied integer. It will be passed back in the response by the server, unmodified. It is useful for matching request and response between the client and server.

**get\_bytes** ()

Serialize the message

**Returns** Serialized message

**Return type** `bytearray`

**class** `pykafka.protocol.Response`

Bases: `object`

Base class for Response objects.

**\_\_weakref\_\_**

list of weak references to the object (if defined)

**raise\_error** (*err\_code, response*)

Raise an error based on the Kafka error code

**Parameters**

- **err\_code** – The error code from Kafka
- **response** – The unpacked raw data from the response

**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`

**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*)

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
```

### Variables

- **messages** – The list of messages currently in the `MessageSet`
- **compression\_type** – compression to use for the messages

**\_\_init\_\_** (*compression\_type=0*, *messages=None*)

Create a new `MessageSet`

**Parameters**

- **compression\_type** – Compression to use on the messages
- **messages** – An initial list of messages for the set

**\_\_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

**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='')
```

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, off-
         sets_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='')

```

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** (*bytes*) – 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 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 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*) – The generation id with which to make group requests
- **consumer\_id** (*bytes*) – The identifying string to use for this consumer on group requests

**\_\_iter\_\_** ()

Yield an infinite stream of messages until the consumer times out

**\_\_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** ()

Commit offsets for this consumer’s partitions

Uses the offset commit/fetch API

**consume** (*block=True*)

Get one message from the consumer.

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



**fetch ()**

Fetch new messages for all partitions

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

Issue an `OffsetRequest` for each partition and set the appropriate returned offset in the consumer's internal offset counter.

**Parameters** `partition_offsets` (Sequence of tuples of the form (`pykafka.partition.Partition`, int)) – (`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 of the message whose offset the partition should have OR the new “most recently consumed” offset the partition should have

NOTE: If an instance of `timestamp_or_offset` is treated by kafka as an invalid offset timestamp, this function directly sets the consumer's internal offset counter for that partition to that instance of `timestamp_or_offset`. This counter represents the offset most recently consumed. On the next fetch request, the consumer attempts to fetch messages starting from that offset plus one. See the following link for more information on what kafka treats as a valid offset timestamp: <https://cwiki.apache.org/confluence/display/KAFKA/A+Guide+To+The+Kafka+Protocol#AGuideToTheKafkaProtocol-OffsetRequest>

**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

## pykafka.topic

Author: Keith Bourgoïn, 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.

**\_\_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 earliest or latest offset.

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

**Parameters**

- **offsets\_before** (*int*) – Return an offset from before this timestamp (in milliseconds)
- **max\_offsets** (*int*) – The maximum number of offsets to return

**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

## pykafka.utils.compression

Author: Keith Bourgoïn

`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.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

## pykafka.utils.socket

Author: Keith Bourgoïn, 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*

## pykafka.utils.struct\_helpers

Author: Keith Bourgoïn, Emmett Butler

pykafka.utils.struct\_helpers.**unpack\_from** (*fmt, buff, offset=0*)  
A customized version of *struct.unpack\_from*

This is a convenience 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:

- *W* for strings (int16 followed by byte array)
- *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.

## Indices and tables

- genindex
- modindex
- search

### p

- `pykafka.balancedconsumer`, 20
- `pykafka.broker`, 24
- `pykafka.client`, 27
- `pykafka.cluster`, 29
- `pykafka.common`, 30
- `pykafka.connection`, 31
- `pykafka.exceptions`, 33
- `pykafka.handlers`, 36
- `pykafka.managedbalancedconsumer`, 40
- `pykafka.partition`, 43
- `pykafka.partitioners`, 44
- `pykafka.producer`, 45
- `pykafka.protocol`, 48
- `pykafka.simpleconsumer`, 58
- `pykafka.topic`, 61
- `pykafka.utils.compression`, 63
- `pykafka.utils.error_handlers`, 63
- `pykafka.utils.socket`, 64
- `pykafka.utils.struct_helpers`, 64



## Symbols

- `__call__()` (pykafka.partitioners.GroupHashingPartitioner method), 45
- `__call__()` (pykafka.partitioners.HashingPartitioner method), 44
- `__del__()` (pykafka.connection.BrokerConnection method), 32
- `__del__()` (pykafka.simpleconsumer.SimpleConsumer method), 58
- `__enter__()` (pykafka.handlers.ThreadingHandler.Semaphore method), 37
- `__enter__()` (pykafka.producer.Producer method), 45
- `__exit__()` (pykafka.producer.Producer method), 45
- `__getnewargs__()` (pykafka.handlers.RequestHandler.Shared method), 38
- `__getnewargs__()` (pykafka.handlers.RequestHandler.Task method), 39
- `__getstate__()` (pykafka.handlers.RequestHandler.Shared method), 38
- `__getstate__()` (pykafka.handlers.RequestHandler.Task method), 39
- `__init__()` (pykafka.balancedconsumer.BalancedConsumer method), 20
- `__init__()` (pykafka.broker.Broker method), 24
- `__init__()` (pykafka.client.KafkaClient method), 28
- `__init__()` (pykafka.cluster.Cluster method), 29
- `__init__()` (pykafka.connection.BrokerConnection method), 32
- `__init__()` (pykafka.connection.SslConfig method), 31
- `__init__()` (pykafka.handlers.RequestHandler method), 39
- `__init__()` (pykafka.handlers.ResponseFuture method), 36
- `__init__()` (pykafka.managedbalancedconsumer.ManagedBalancedConsumer method), 40
- `__init__()` (pykafka.partition.Partition method), 43
- `__init__()` (pykafka.partitioners.GroupHashingPartitioner method), 45
- `__init__()` (pykafka.partitioners.HashingPartitioner method), 44
- `__init__()` (pykafka.producer.Producer method), 45
- `__init__()` (pykafka.protocol.FetchRequest method), 52
- `__init__()` (pykafka.protocol.FetchResponse method), 53
- `__init__()` (pykafka.protocol.GroupCoordinatorRequest method), 55
- `__init__()` (pykafka.protocol.GroupCoordinatorResponse method), 55
- `__init__()` (pykafka.protocol.MessageSet method), 57
- `__init__()` (pykafka.protocol.MetadataRequest method), 48
- `__init__()` (pykafka.protocol.MetadataResponse method), 49
- `__init__()` (pykafka.protocol.OffsetCommitRequest method), 51
- `__init__()` (pykafka.protocol.OffsetCommitResponse method), 53
- `__init__()` (pykafka.protocol.OffsetFetchRequest method), 54
- `__init__()` (pykafka.protocol.OffsetFetchResponse method), 54
- `__init__()` (pykafka.protocol.OffsetRequest method), 50
- `__init__()` (pykafka.protocol.OffsetResponse method), 51
- `__init__()` (pykafka.protocol.ProduceRequest method), 49
- `__init__()` (pykafka.protocol.ProduceResponse method), 50
- `__init__()` (pykafka.simpleconsumer.SimpleConsumer method), 58
- `__init__()` (pykafka.topic.Topic method), 61
- `__iter__()` (pykafka.balancedconsumer.BalancedConsumer method), 22
- `__iter__()` (pykafka.simpleconsumer.SimpleConsumer method), 60
- `__len__()` (pykafka.protocol.FetchRequest method), 52
- `__len__()` (pykafka.protocol.GroupCoordinatorRequest method), 55
- `__len__()` (pykafka.protocol.MessageSet method), 58
- `__len__()` (pykafka.protocol.MetadataRequest method), 48

\_\_len\_\_() (pykafka.protocol.OffsetCommitRequest method), 51  
 \_\_len\_\_() (pykafka.protocol.OffsetFetchRequest method), 54  
 \_\_len\_\_() (pykafka.protocol.OffsetRequest method), 50  
 \_\_len\_\_() (pykafka.protocol.ProduceRequest method), 49  
 \_\_new\_\_() (pykafka.handlers.RequestHandler.Shared static method), 38  
 \_\_new\_\_() (pykafka.handlers.RequestHandler.Task static method), 39  
 \_\_repr\_\_() (pykafka.handlers.RequestHandler.Shared method), 39  
 \_\_repr\_\_() (pykafka.handlers.RequestHandler.Task method), 39  
 \_\_weakref\_\_ (pykafka.balancedconsumer.BalancedConsumer attribute), 22  
 \_\_weakref\_\_ (pykafka.broker.Broker attribute), 25  
 \_\_weakref\_\_ (pykafka.client.KafkaClient attribute), 29  
 \_\_weakref\_\_ (pykafka.cluster.Cluster attribute), 29  
 \_\_weakref\_\_ (pykafka.common.CompressionType attribute), 31  
 \_\_weakref\_\_ (pykafka.common.OffsetType attribute), 31  
 \_\_weakref\_\_ (pykafka.connection.BrokerConnection attribute), 32  
 \_\_weakref\_\_ (pykafka.connection.SslConfig attribute), 32  
 \_\_weakref\_\_ (pykafka.exceptions.KafkaException attribute), 33  
 \_\_weakref\_\_ (pykafka.handlers.Handler attribute), 36  
 \_\_weakref\_\_ (pykafka.handlers.RequestHandler attribute), 39  
 \_\_weakref\_\_ (pykafka.handlers.ResponseFuture attribute), 36  
 \_\_weakref\_\_ (pykafka.handlers.ThreadingHandler.Semaphore attribute), 38  
 \_\_weakref\_\_ (pykafka.partition.Partition attribute), 43  
 \_\_weakref\_\_ (pykafka.partitioners.BasePartitioner attribute), 44  
 \_\_weakref\_\_ (pykafka.producer.Producer attribute), 47  
 \_\_weakref\_\_ (pykafka.protocol.MessageSet attribute), 58  
 \_\_weakref\_\_ (pykafka.protocol.Request attribute), 56  
 \_\_weakref\_\_ (pykafka.protocol.Response attribute), 56  
 \_\_weakref\_\_ (pykafka.simpleconsumer.SimpleConsumer attribute), 60  
 \_\_weakref\_\_ (pykafka.topic.Topic attribute), 62  
 \_add\_partitions() (pykafka.balancedconsumer.BalancedConsumer method), 22  
 \_add\_self() (pykafka.balancedconsumer.BalancedConsumer method), 22  
 \_asdict() (pykafka.handlers.RequestHandler.Shared method), 39  
 \_asdict() (pykafka.handlers.RequestHandler.Task method), 39  
 \_auto\_commit() (pykafka.simpleconsumer.SimpleConsumer method), 60  
 \_build\_default\_error\_handlers() (pykafka.managedbalancedconsumer.ManagedBalancedConsumer method), 42  
 \_build\_default\_error\_handlers() (pykafka.simpleconsumer.SimpleConsumer method), 60  
 \_build\_watch\_callback() (pykafka.balancedconsumer.BalancedConsumer method), 22  
 \_decide\_partitions() (pykafka.balancedconsumer.BalancedConsumer method), 22  
 \_discover\_group\_coordinator() (pykafka.simpleconsumer.SimpleConsumer method), 60  
 \_get\_brokers\_from\_zookeeper() (pykafka.cluster.Cluster method), 29  
 \_get\_compressed() (pykafka.protocol.MessageSet method), 58  
 \_get\_held\_partitions() (pykafka.balancedconsumer.BalancedConsumer method), 22  
 \_get\_internal\_consumer() (pykafka.balancedconsumer.BalancedConsumer method), 22  
 \_get\_metadata() (pykafka.cluster.Cluster method), 30  
 \_get\_participants() (pykafka.balancedconsumer.BalancedConsumer method), 22  
 \_get\_unique\_req\_handler() (pykafka.broker.Broker method), 25  
 \_handle\_error() (pykafka.managedbalancedconsumer.ManagedBalancedConsumer method), 42  
 \_join\_group() (pykafka.managedbalancedconsumer.ManagedBalancedConsumer method), 42  
 \_legacy\_wrap\_socket() (pykafka.connection.SslConfig method), 32  
 \_make() (pykafka.handlers.RequestHandler.Shared class method), 39  
 \_make() (pykafka.handlers.RequestHandler.Task class method), 39  
 \_partitions (pykafka.balancedconsumer.BalancedConsumer attribute), 22  
 \_path\_from\_partition() (pykafka.balancedconsumer.BalancedConsumer method), 23  
 \_path\_self (pykafka.balancedconsumer.BalancedConsumer attribute), 23  
 \_produce() (pykafka.producer.Producer method), 47  
 \_raise\_worker\_exceptions() (pykafka.balancedconsumer.BalancedConsumer method), 23  
 \_raise\_worker\_exceptions() (pykafka.producer.Producer method), 47  
 \_raise\_worker\_exceptions() (pykafka.simpleconsumer.SimpleConsumer method), 60  
 \_rebalance() (pykafka.balancedconsumer.BalancedConsumer



method), 23

`_remove_partitions()` (pykafka.balancedconsumer.BalancedConsumer method), 23

`_replace()` (pykafka.handlers.RequestHandler.Shared method), 39

`_replace()` (pykafka.handlers.RequestHandler.Task method), 39

`_request_metadata()` (pykafka.cluster.Cluster method), 30

`_send_request()` (pykafka.producer.Producer method), 47

`_set_watches()` (pykafka.balancedconsumer.BalancedConsumer method), 23

`_setup_autocommit_worker()` (pykafka.simpleconsumer.SimpleConsumer method), 60

`_setup_fetch_workers()` (pykafka.simpleconsumer.SimpleConsumer method), 60

`_setup_heartbeat_worker()` (pykafka.managedbalancedconsumer.ManagedBalancedConsumer method), 42

`_setup_internal_consumer()` (pykafka.balancedconsumer.BalancedConsumer method), 23

`_setup_owned_brokers()` (pykafka.producer.Producer method), 47

`_setup_zookeeper()` (pykafka.balancedconsumer.BalancedConsumer method), 23

`_start_thread()` (pykafka.handlers.RequestHandler method), 39

`_sync_group()` (pykafka.managedbalancedconsumer.ManagedBalancedConsumer method), 42

`_unpack_message_set()` (pykafka.protocol.FetchResponse method), 53

`_update()` (pykafka.producer.Producer method), 47

`_update()` (pykafka.simpleconsumer.SimpleConsumer method), 60

`_update_brokers()` (pykafka.cluster.Cluster method), 30

`_update_member_assignment()` (pykafka.balancedconsumer.BalancedConsumer method), 23

`_update_member_assignment()` (pykafka.managedbalancedconsumer.ManagedBalancedConsumer method), 42

`_wait_all()` (pykafka.producer.Producer method), 47

`_wait_for_slot_available()` (pykafka.simpleconsumer.SimpleConsumer method), 60

`_write_header()` (pykafka.protocol.Request method), 56

`add_request()` (pykafka.protocol.FetchRequest method), 52

`API_KEY` (pykafka.protocol.FetchRequest attribute), 52

`API_KEY` (pykafka.protocol.GroupCoordinatorRequest attribute), 55

`API_KEY` (pykafka.protocol.MetadataRequest attribute), 48

`API_KEY` (pykafka.protocol.OffsetCommitRequest attribute), 51

`API_KEY` (pykafka.protocol.OffsetFetchRequest attribute), 54

`API_KEY` (pykafka.protocol.OffsetRequest attribute), 50

`API_KEY` (pykafka.protocol.ProduceRequest attribute), 49

`API_KEY` (pykafka.protocol.Request method), 56

## B

`BasePartitioner` (class in pykafka.partitioners), 44

`Broker` (class in pykafka.broker), 24

`BrokerConnection` (class in pykafka.connection), 32

`brokers` (pykafka.cluster.Cluster attribute), 30

## C

`Cluster` (class in pykafka.cluster), 29

`commit_consumer_group_offsets()` (pykafka.broker.Broker method), 25

`commit_offsets()` (pykafka.balancedconsumer.BalancedConsumer method), 23

`commit_offsets()` (pykafka.simpleconsumer.SimpleConsumer method), 60

`CompressionType` (class in pykafka.common), 31

`connect()` (pykafka.broker.Broker method), 25

`connect()` (pykafka.connection.BrokerConnection method), 32

`connect_offsets_channel()` (pykafka.broker.Broker method), 25

`connected` (pykafka.broker.Broker attribute), 25

`connected` (pykafka.connection.BrokerConnection attribute), 32

`connection` (pykafka.handlers.RequestHandler.Shared attribute), 39

`consume()` (pykafka.balancedconsumer.BalancedConsumer method), 23

`consume()` (pykafka.simpleconsumer.SimpleConsumer method), 60

`ConsumerStoppedException`, 33

## D

`decode()` (pykafka.protocol.MessageSet class method), 58

`decode_gzip()` (in module pykafka.utils.compression), 63

`decode_snappy()` (in module pykafka.utils.compression), 63

- disconnect() (pykafka.connection.BrokerConnection method), 32
- ## E
- earliest\_available\_offset() (pykafka.partition.Partition method), 43
- earliest\_available\_offsets() (pykafka.topic.Topic method), 62
- empty() (pykafka.handlers.ThreadingHandler.Queue method), 36
- encode\_gzip() (in module pykafka.utils.compression), 63
- encode\_snappy() (in module pykafka.utils.compression), 63
- ending (pykafka.handlers.RequestHandler.Shared attribute), 39
- Event() (pykafka.handlers.ThreadingHandler method), 36
- ## F
- fetch() (pykafka.simpleconsumer.SimpleConsumer method), 60
- fetch\_consumer\_group\_offsets() (pykafka.broker.Broker method), 25
- fetch\_offset\_limit() (pykafka.partition.Partition method), 43
- fetch\_offset\_limits() (pykafka.topic.Topic method), 62
- fetch\_offsets() (pykafka.simpleconsumer.SimpleConsumer method), 61
- FetchRequest (class in pykafka.protocol), 51
- FetchResponse (class in pykafka.protocol), 52
- from\_metadata() (pykafka.broker.Broker class method), 26
- full() (pykafka.handlers.ThreadingHandler.Queue method), 36
- future (pykafka.handlers.RequestHandler.Task attribute), 39
- ## G
- get() (pykafka.handlers.ResponseFuture method), 36
- get() (pykafka.handlers.ThreadingHandler.Queue method), 36
- get\_balanced\_consumer() (pykafka.topic.Topic method), 62
- get\_bytes() (pykafka.protocol.FetchRequest method), 52
- get\_bytes() (pykafka.protocol.GroupCoordinatorRequest method), 55
- get\_bytes() (pykafka.protocol.MetadataRequest method), 48
- get\_bytes() (pykafka.protocol.OffsetCommitRequest method), 51
- get\_bytes() (pykafka.protocol.OffsetFetchRequest method), 54
- get\_bytes() (pykafka.protocol.OffsetRequest method), 50
- get\_bytes() (pykafka.protocol.ProduceRequest method), 50
- get\_bytes() (pykafka.protocol.Request method), 56
- get\_delivery\_report() (pykafka.producer.Producer method), 47
- get\_group\_coordinator() (pykafka.cluster.Cluster method), 30
- get\_managed\_group\_descriptions() (pykafka.cluster.Cluster method), 30
- get\_nowait() (pykafka.handlers.ThreadingHandler.Queue method), 37
- get\_producer() (pykafka.topic.Topic method), 62
- get\_simple\_consumer() (pykafka.topic.Topic method), 62
- get\_subclass() (pykafka.protocol.FetchResponse static method), 53
- get\_sync\_producer() (pykafka.topic.Topic method), 62
- GroupAuthorizationFailed, 33
- GroupCoordinatorNotAvailable, 33
- GroupCoordinatorRequest (class in pykafka.protocol), 54
- GroupCoordinatorResponse (class in pykafka.protocol), 55
- GroupHashingPartitioner (class in pykafka.partitioners), 45
- GroupLoadInProgress, 33
- ## H
- handle\_partition\_responses() (in module pykafka.utils.error\_handlers), 63
- Handler (class in pykafka.handlers), 36
- handler (pykafka.broker.Broker attribute), 26
- handler (pykafka.cluster.Cluster attribute), 30
- HashingPartitioner (class in pykafka.partitioners), 44
- heartbeat() (pykafka.broker.Broker method), 26
- held\_offsets (pykafka.balancedconsumer.BalancedConsumer attribute), 23
- held\_offsets (pykafka.simpleconsumer.SimpleConsumer attribute), 61
- host (pykafka.broker.Broker attribute), 26
- ## I
- id (pykafka.broker.Broker attribute), 26
- id (pykafka.partition.Partition attribute), 43
- IllegalGeneration, 33
- InconsistentGroupProtocol, 33
- InvalidMessageError, 33
- InvalidMessageSize, 33
- InvalidSessionTimeout, 33
- isr (pykafka.partition.Partition attribute), 43
- ## J
- join() (pykafka.handlers.ThreadingHandler.Queue method), 37
- join\_group() (pykafka.broker.Broker method), 26
- ## K
- KafkaClient (class in pykafka.client), 27

KafkaException, 33

## L

latest\_available\_offset() (pykafka.partition.Partition method), 43

latest\_available\_offsets() (pykafka.topic.Topic method), 62

leader (pykafka.partition.Partition attribute), 43

LeaderNotAvailable, 34

leave\_group() (pykafka.broker.Broker method), 27

Lock() (pykafka.handlers.ThreadingHandler method), 36

## M

ManagedBalancedConsumer (class in pykafka.managedbalancedconsumer), 40

Message (class in pykafka.common), 30

Message (class in pykafka.protocol), 56

message\_count() (pykafka.protocol.ProduceRequest method), 50

messages (pykafka.protocol.ProduceRequest attribute), 50

MessageSet (class in pykafka.protocol), 57

MessageSizeTooLarge, 34

MetadataRequest (class in pykafka.protocol), 48

MetadataResponse (class in pykafka.protocol), 48

## N

name (pykafka.topic.Topic attribute), 62

NoBrokersAvailableError, 34

NoMessagesConsumedError, 34

NotCoordinatorForGroup, 34

NotLeaderForPartition, 34

## O

OffsetCommitRequest (class in pykafka.protocol), 51

OffsetCommitResponse (class in pykafka.protocol), 53

OffsetFetchRequest (class in pykafka.protocol), 53

OffsetFetchResponse (class in pykafka.protocol), 54

OffsetMetadataTooLarge, 34

OffsetOutOfRangeError, 34

OffsetRequest (class in pykafka.protocol), 50

OffsetRequestFailedError, 34

OffsetResponse (class in pykafka.protocol), 51

offsets\_channel\_connected (pykafka.broker.Broker attribute), 27

offsets\_channel\_handler (pykafka.broker.Broker attribute), 27

OffsetType (class in pykafka.common), 31

## P

pack\_into() (pykafka.protocol.Message method), 57

pack\_into() (pykafka.protocol.MessageSet method), 58

Partition (class in pykafka.partition), 43

PartitionFetchRequest (class in pykafka.protocol), 53

PartitionOffsetCommitRequest (class in pykafka.protocol), 55

PartitionOffsetFetchRequest (class in pykafka.protocol), 55

PartitionOffsetRequest (class in pykafka.protocol), 54

PartitionOwnedError, 34

partitions (pykafka.balancedconsumer.BalancedConsumer attribute), 23

partitions (pykafka.simpleconsumer.SimpleConsumer attribute), 61

partitions (pykafka.topic.Topic attribute), 62

port (pykafka.broker.Broker attribute), 27

produce() (pykafka.producer.Producer method), 48

ProduceFailureError, 34

Producer (class in pykafka.producer), 45

ProduceRequest (class in pykafka.protocol), 49

ProduceResponse (class in pykafka.protocol), 50

ProducerQueueFullError, 34

ProducerStoppedException, 34

ProtocolClientError, 35

put() (pykafka.handlers.ThreadingHandler.Queue method), 37

put\_nowait() (pykafka.handlers.ThreadingHandler.Queue method), 37

pykafka.balancedconsumer (module), 20

pykafka.broker (module), 24

pykafka.client (module), 27

pykafka.cluster (module), 29

pykafka.common (module), 30

pykafka.connection (module), 31

pykafka.exceptions (module), 33

pykafka.handlers (module), 36

pykafka.managedbalancedconsumer (module), 40

pykafka.partition (module), 43

pykafka.partitioners (module), 44

pykafka.producer (module), 45

pykafka.protocol (module), 48

pykafka.simpleconsumer (module), 58

pykafka.topic (module), 61

pykafka.utils.compression (module), 63

pykafka.utils.error\_handlers (module), 63

pykafka.utils.socket (module), 64

pykafka.utils.struct\_helpers (module), 64

## Q

qsize() (pykafka.handlers.ThreadingHandler.Queue method), 37

## R

raise\_error() (in module pykafka.utils.error\_handlers), 64

raise\_error() (pykafka.protocol.Response method), 56

random\_partitioner() (in module pykafka.partitioners), 44

RdKafkaException, 35

- RdKafkaStoppedException, 35
  - RebalanceInProgress, 35
  - reconnect() (pykafka.connection.BrokerConnection method), 32
  - recvall\_into() (in module pykafka.utils.socket), 64
  - release() (pykafka.handlers.ThreadingHandler.Semaphore method), 38
  - replicas (pykafka.partition.Partition attribute), 44
  - Request (class in pykafka.protocol), 56
  - request (pykafka.handlers.RequestHandler.Task attribute), 39
  - request() (pykafka.connection.BrokerConnection method), 32
  - request() (pykafka.handlers.RequestHandler method), 39
  - RequestHandler (class in pykafka.handlers), 38
  - RequestHandler.Shared (class in pykafka.handlers), 38
  - RequestHandler.Task (class in pykafka.handlers), 39
  - requests (pykafka.handlers.RequestHandler.Shared attribute), 39
  - RequestTimeout, 35
  - reset\_offsets() (pykafka.balancedconsumer.BalancedConsumer method), 23
  - reset\_offsets() (pykafka.simpleconsumer.SimpleConsumer method), 61
  - Response (class in pykafka.protocol), 56
  - response() (pykafka.connection.BrokerConnection method), 32
  - ResponseFuture (class in pykafka.handlers), 36
- S**
- set\_error() (pykafka.handlers.ResponseFuture method), 36
  - set\_response() (pykafka.handlers.ResponseFuture method), 36
  - SimpleConsumer (class in pykafka.simpleconsumer), 58
  - Socket (pykafka.handlers.ThreadingHandler attribute), 38
  - SocketDisconnectedError, 35
  - spawn() (pykafka.handlers.Handler method), 36
  - SslConfig (class in pykafka.connection), 31
  - start() (pykafka.balancedconsumer.BalancedConsumer method), 24
  - start() (pykafka.handlers.RequestHandler method), 40
  - start() (pykafka.managedbalancedconsumer.ManagedBalancedConsumer method), 42
  - start() (pykafka.producer.Producer method), 48
  - start() (pykafka.simpleconsumer.SimpleConsumer method), 61
  - stop() (pykafka.balancedconsumer.BalancedConsumer method), 24
  - stop() (pykafka.handlers.RequestHandler method), 40
  - stop() (pykafka.managedbalancedconsumer.ManagedBalancedConsumer method), 43
  - stop() (pykafka.producer.Producer method), 48
  - stop() (pykafka.simpleconsumer.SimpleConsumer method), 61
  - sync\_group() (pykafka.broker.Broker method), 27
- T**
- task\_done() (pykafka.handlers.ThreadingHandler.Queue method), 37
  - ThreadingHandler (class in pykafka.handlers), 36
  - ThreadingHandler.Queue (class in pykafka.handlers), 36
  - ThreadingHandler.Semaphore (class in pykafka.handlers), 37
  - timestamp\_dt (pykafka.protocol.Message attribute), 57
  - Topic (class in pykafka.topic), 61
  - topic (pykafka.balancedconsumer.BalancedConsumer attribute), 24
  - topic (pykafka.partition.Partition attribute), 44
  - topic (pykafka.simpleconsumer.SimpleConsumer attribute), 61
  - TopicAuthorizationFailed, 35
  - topics (pykafka.cluster.Cluster attribute), 30
- U**
- UnknownError, 35
  - UnknownMemberId, 35
  - UnknownTopicOrPartition, 35
  - unpack\_from() (in module pykafka.utils.struct\_helpers), 64
  - update() (pykafka.cluster.Cluster method), 30
  - update() (pykafka.partition.Partition method), 44
  - update() (pykafka.topic.Topic method), 62
  - update\_cluster() (pykafka.client.KafkaClient method), 29
- W**
- wrap\_socket() (pykafka.connection.SslConfig method), 32