rabbitpy Documentation

Release 0.26.2

Gavin M. Roy

October 28, 2016

Contents

rabbitpy is a pure python, thread-safe [1], and pythonic BSD Licensed AMQP/RabbitMQ library that supports Python 2.6+ and 3.2+. rabbitpy aims to provide a simple and easy to use API for interfacing with RabbitMQ, minimizing the programming overhead often found in other libraries.

[1] If you're looking to use rabbitpy in a multi-threaded application, you should the notes about multi-threaded use in `threads`.

CHAPTER 1

Installation

rabbitpy is available from the Python Package Index and can be installed by running **`easy_install rabbitpy`** or **`pip install rabbitpy`**

CHAPTER 2

API Documentation

rabbitpy is designed to have as simple and pythonic of an API as possible while still remaining true to RabbitMQ and to the AMQP 0-9-1 specification. There are two basic ways to interact with rabbitpy, using the simple wrapper methods:

2.1 Simple API Methods

rabbitpy's simple API methods are meant for one off use, either in your apps or in the python interpreter. For example, if your application publishes a single message as part of its lifetime, `rabbitpy.publish()` should be enough for almost any publishing concern. However if you are publishing more than one message, it is not an efficient method to use as it connects and disconnects from RabbitMQ on each invocation. `rabbitpy.get()` also connects and disconnects on each invocation. `rabbitpy.consume()` does stay connected as long as you're iterating through the messages returned by it. Exiting the generator will close the connection. For a more complete api, see the rabbitpy core API. Wrapper methods for easy access to common operations, making them both less complex and less verbose for one off or simple use cases.

`rabbitpy.simple.`**`consume`**(*uri=None*, *queue_name=None*, *no_ack=False*, *prefetch=None*, *priority=None*)

Consume messages from the queue as a generator:

```
for message in rabbitpy.consume('amqp://localhost/%2F', 'my_queue'):
    message.ack()
```

Parameters

- **`uri`** (*str*) – AMQP connection URI
- **`queue_name`** (*str*) – The name of the queue to consume from
- **`no_ack`** (*bool*) – Do not require acknowledgements
- **`prefetch`** (*int*) – Set a prefetch count for the channel
- **`priority`** (*int*) – Set the consumer priority

Return type `Iterator`

Raises py:class:*ValueError*

`rabbitpy.simple.`**`create_direct_exchange`**(*uri=None*, *exchange_name=None*, *durable=True*)

Create a direct exchange with RabbitMQ. This should only be used for one-off operations.

Parameters

- **uri** (*str*) – AMQP URI to connect to
- **exchange_name** (*str*) – The exchange name to create
- **durable** (*bool*) – Exchange should survive server restarts

Raises ValueError

Raises rabbitpy.RemoteClosedException

rabbitpy.simple.**create_fanout_exchange**(*uri=None*, *exchange_name=None*, *durable=True*)
Create a fanout exchange with RabbitMQ. This should only be used for one-off operations.

Parameters

- **uri** (*str*) – AMQP URI to connect to
- **exchange_name** (*str*) – The exchange name to create
- **durable** (*bool*) – Exchange should survive server restarts

Raises ValueError

Raises rabbitpy.RemoteClosedException

rabbitpy.simple.**create_headers_exchange**(*uri=None*, *exchange_name=None*, *durable=True*)
Create a headers exchange with RabbitMQ. This should only be used for one-off operations.

Parameters

- **uri** (*str*) – AMQP URI to connect to
- **exchange_name** (*str*) – The exchange name to create
- **durable** (*bool*) – Exchange should survive server restarts

Raises ValueError

Raises rabbitpy.RemoteClosedException

rabbitpy.simple.**create_queue**(*uri=None*, *queue_name=''*, *durable=True*, *auto_delete=False*, *max_length=None*, *message_ttl=None*, *expires=None*, *dead_letter_exchange=None*, *dead_letter_routing_key=None*, *arguments=None*)
Create a queue with RabbitMQ. This should only be used for one-off operations. If a queue name is omitted, the name will be automatically generated by RabbitMQ.

Parameters

- **uri** (*str*) – AMQP URI to connect to
- **queue_name** (*str*) – The queue name to create
- **durable** (*bool*) – Indicates if the queue should survive a RabbitMQ is restart
- **auto_delete** (*bool*) – Automatically delete when all consumers disconnect
- **max_length** (*int*) – Maximum queue length
- **message_ttl** (*int*) – Time-to-live of a message in milliseconds
- **expires** (*int*) – Milliseconds until a queue is removed after becoming idle
- **dead_letter_exchange** (*str*) – Dead letter exchange for rejected messages
- **dead_letter_routing_key** (*str*) – Routing key for dead lettered messages
- **arguments** (*dict*) – Custom arguments for the queue

Raises ValueError

Raises rabbitpy.RemoteClosedException

rabbitpy.simple.**create_topic_exchange**(*uri=None*, *exchange_name=None*, *durable=True*)
Create an exchange from RabbitMQ. This should only be used for one-off operations.

Parameters

- **uri** (*str*) – AMQP URI to connect to
- **exchange_name** (*str*) – The exchange name to create
- **durable** (*bool*) – Exchange should survive server restarts

Raises ValueError

Raises rabbitpy.RemoteClosedException

rabbitpy.simple.**delete_exchange**(*uri=None*, *exchange_name=None*)
Delete an exchange from RabbitMQ. This should only be used for one-off operations.

Parameters

- **uri** (*str*) – AMQP URI to connect to
- **exchange_name** (*str*) – The exchange name to delete

Raises ValueError

Raises rabbitpy.RemoteClosedException

rabbitpy.simple.**delete_queue**(*uri=None*, *queue_name=None*)
Delete a queue from RabbitMQ. This should only be used for one-off operations.

Parameters

- **uri** (*str*) – AMQP URI to connect to
- **queue_name** (*str*) – The queue name to delete

Return type bool

Raises ValueError

Raises rabbitpy.RemoteClosedException

rabbitpy.simple.**get**(*uri=None*, *queue_name=None*)
Get a message from RabbitMQ, auto-acknowledging with RabbitMQ if one is returned.

Invoke directly as rabbitpy.get()

Parameters

- **uri** (*str*) – AMQP URI to connect to
- **queue_name** (*str*) – The queue name to get the message from

Return type py:class:*rabbitpy.message.Message* or None

Raises py:class:*ValueError*

rabbitpy.simple.**publish**(*uri=None*, *exchange_name=None*, *routing_key=None*, *body=None*, *properties=None*, *confirm=False*)
Publish a message to RabbitMQ. This should only be used for one-off publishing, as you will suffer a performance penalty if you use it repeatedly instead creating a connection and channel and publishing on that

Parameters

- **uri** (*str*) – AMQP URI to connect to
- **exchange_name** (*str*) – The exchange to publish to

- **routing_key** (*str*) – The routing_key to publish with
- **or unicode or bytes or dict or list** (*str*) – The message body
- **properties** (*dict*) – Dict representation of Basic.Properties
- **confirm** (*bool*) – Confirm this delivery with Publisher Confirms

Return type bool or None

And by using the core objects:

2.2 AMQP Adapter

While the core rabbitpy API strives to provide an easy to use, Pythonic interface for RabbitMQ, some developers may prefer a less opinionated AMQP interface. The *rabbitpy.AMQP* adapter provides a more traditional AMQP client library API seen in libraries like pika.

New in version 0.26.

2.2.1 Example

The following example will connect to RabbitMQ and use the *rabbitpy.AMQP* adapter to consume and acknowledge messages.

```
import rabbitpy

with rabbitpy.Connection() as conn:
    with conn.channel() as channel:
        amqp = rabbitpy.AMQP(channel)

        for message in amqp.basic_consume('queue-name'):
            print(message)
```

2.2.2 API Documentation

class rabbitpy.**AMQP**(*channel*)

The AMQP Adapter provides a more generic, non-opinionated interface to RabbitMQ by providing methods that map to the AMQP API.

Parameters **channel** (*rabbitmq.channel.Channel*) – The channel to use

basic_ack(*delivery_tag=0*, *multiple=False*)

Acknowledge one or more messages

This method acknowledges one or more messages delivered via the Deliver or Get-Ok methods. The client can ask to confirm a single message or a set of messages up to and including a specific message.

Parameters

- **delivery_tag** (*int/long*) – Server-assigned delivery tag
- **multiple** (*bool*) – Acknowledge multiple messages

basic_cancel(*consumer_tag=''*, *nowait=False*)

End a queue consumer

This method cancels a consumer. This does not affect already delivered messages, but it does mean the server will not send any more messages for that consumer. The client may receive an arbitrary number of messages in between sending the cancel method and receiving the cancel- ok reply.

Parameters

- **consumer_tag** (*str*) – Consumer tag
- **nowait** (*bool*) – Do not send a reply method

basic_consume(*queue=''*, *consumer_tag=''*, *no_local=False*, *no_ack=False*, *exclusive=False*, *nowait=False*, *arguments=None*)

Start a queue consumer

This method asks the server to start a "consumer", which is a transient request for messages from a specific queue. Consumers last as long as the channel they were declared on, or until the client cancels them.

This method will act as an generator, returning messages as they are delivered from the server.

Example use:

```
for message in basic_consume(queue_name):
    print message.body
    message.ack()
```

Parameters

- **queue** (*str*) – The queue name to consume from
- **consumer-tag** (*str*) – The consumer tag
- **no_local** (*bool*) – Do not deliver own messages
- **no_ack** (*bool*) – No acknowledgement needed
- **exclusive** (*bool*) – Request exclusive access
- **nowait** (*bool*) – Do not send a reply method
- **arguments** (*dict*) – Arguments for declaration

basic_get(*queue=''*, *no_ack=False*)

Direct access to a queue

This method provides a direct access to the messages in a queue using a synchronous dialogue that is designed for specific types of application where synchronous functionality is more important than performance.

Parameters

- **queue** (*str*) – The queue name
- **no_ack** (*bool*) – No acknowledgement needed

basic_nack(*delivery_tag=0*, *multiple=False*, *requeue=True*)

Reject one or more incoming messages.

This method allows a client to reject one or more incoming messages. It can be used to interrupt and cancel large incoming messages, or return untreatable messages to their original queue. This method is also used by the server to inform publishers on channels in confirm mode of unhandled messages. If a publisher receives this method, it probably needs to republish the offending messages.

Parameters

- **delivery_tag** (*int/long*) – Server-assigned delivery tag

- **multiple** (*bool*) – Reject multiple messages
- **requeue** (*bool*) – Requeue the message

basic_publish(*exchange=''*, *routing_key=''*, *body=''*, *properties=None*, *mandatory=False*, *immediate=False*)
Publish a message

This method publishes a message to a specific exchange. The message will be routed to queues as defined by the exchange configuration and distributed to any active consumers when the transaction, if any, is committed.

Parameters

- **exchange** (*str*) – The exchange name
- **routing_key** (*str*) – Message routing key
- **mandatory** (*bool*) – Indicate mandatory routing
- **immediate** (*bool*) – Request immediate delivery

Returns bool or None

basic_qos(*prefetch_size=0*, *prefetch_count=0*, *global_flag=False*)
Specify quality of service

This method requests a specific quality of service. The QoS can be specified for the current channel or for all channels on the connection. The particular properties and semantics of a qos method always depend on the content class semantics. Though the qos method could in principle apply to both peers, it is currently meaningful only for the server.

Parameters

- **prefetch_size** (*int/long*) – Prefetch window in octets
- **prefetch_count** (*int*) – Prefetch window in messages
- **global_flag** (*bool*) – Apply to entire connection

basic_recover(*requeue=False*)
Redeliver unacknowledged messages

This method asks the server to redeliver all unacknowledged messages on a specified channel. Zero or more messages may be redelivered. This method replaces the asynchronous Recover.

Parameters **requeue** (*bool*) – Requeue the message

basic_reject(*delivery_tag=0*, *requeue=True*)
Reject an incoming message

This method allows a client to reject a message. It can be used to interrupt and cancel large incoming messages, or return untreatable messages to their original queue.

Parameters

- **delivery_tag** (*int/long*) – Server-assigned delivery tag
- **requeue** (*bool*) – Requeue the message

confirm_select()
This method sets the channel to use publisher acknowledgements. The client can only use this method on a non-transactional channel.

exchange_bind(*destination=''*, *source=''*, *routing_key=''*, *nowait=False*, *arguments=None*)
Bind exchange to an exchange.

This method binds an exchange to an exchange.

Parameters

- **destination** (*str*) – The destination exchange name
- **source** (*str*) – The source exchange name
- **routing_key** (*str*) – The routing key to bind with
- **nowait** (*bool*) – Do not send a reply method
- **arguments** (*dict*) – Optional arguments

exchange_declare(*exchange=''*, *exchange_type='direct'*, *passive=False*, *durable=False*, *auto_delete=False*, *internal=False*, *nowait=False*, *arguments=None*)

Verify exchange exists, create if needed

This method creates an exchange if it does not already exist, and if the exchange exists, verifies that it is of the correct and expected class.

Parameters

- **exchange** (*str*) – The exchange name
- **exchange_type** (*str*) – Exchange type
- **passive** (*bool*) – Do not create exchange
- **durable** (*bool*) – Request a durable exchange
- **auto_delete** (*bool*) – Automatically delete when not in use
- **internal** (*bool*) – Deprecated
- **nowait** (*bool*) – Do not send a reply method
- **arguments** (*dict*) – Arguments for declaration

exchange_delete(*exchange=''*, *if_unused=False*, *nowait=False*)

Delete an exchange

This method deletes an exchange. When an exchange is deleted all queue bindings on the exchange are cancelled.

Parameters

- **exchange** (*str*) – The exchange name
- **if_unused** (*bool*) – Delete only if unused
- **nowait** (*bool*) – Do not send a reply method

exchange_unbind(*destination=''*, *source=''*, *routing_key=''*, *nowait=False*, *arguments=None*)

Unbind an exchange from an exchange.

This method unbinds an exchange from an exchange.

Parameters

- **destination** (*str*) – The destination exchange name
- **source** (*str*) – The source exchange name
- **routing_key** (*str*) – The routing key to bind with
- **nowait** (*bool*) – Do not send a reply method
- **arguments** (*dict*) – Optional arguments

queue_bind(*queue=''*, *exchange=''*, *routing_key=''*, *nowait=False*, *arguments=None*)
Bind queue to an exchange

This method binds a queue to an exchange. Until a queue is bound it will not receive any messages. In a classic messaging model, store-and- forward queues are bound to a direct exchange and subscription queues are bound to a topic exchange.

Parameters

- **queue** (*str*) – The queue name
- **exchange** (*str*) – Name of the exchange to bind to
- **routing_key** (*str*) – Message routing key
- **nowait** (*bool*) – Do not send a reply method
- **arguments** (*dict*) – Arguments for binding

queue_declare(*queue=''*, *passive=False*, *durable=False*, *exclusive=False*, *auto_delete=False*, *nowait=False*, *arguments=None*)
Declare queue, create if needed

This method creates or checks a queue. When creating a new queue the client can specify various properties that control the durability of the queue and its contents, and the level of sharing for the queue.

Parameters

- **queue** (*str*) – The queue name
- **passive** (*bool*) – Do not create queue
- **durable** (*bool*) – Request a durable queue
- **exclusive** (*bool*) – Request an exclusive queue
- **auto_delete** (*bool*) – Auto-delete queue when unused
- **nowait** (*bool*) – Do not send a reply method
- **arguments** (*dict*) – Arguments for declaration

queue_delete(*queue=''*, *if_unused=False*, *if_empty=False*, *nowait=False*)
Delete a queue

This method deletes a queue. When a queue is deleted any pending messages are sent to a dead-letter queue if this is defined in the server configuration, and all consumers on the queue are cancelled.

Parameters

- **queue** (*str*) – The queue name
- **if_unused** (*bool*) – Delete only if unused
- **if_empty** (*bool*) – Delete only if empty
- **nowait** (*bool*) – Do not send a reply method

queue_purge(*queue=''*, *nowait=False*)
Purge a queue

This method removes all messages from a queue which are not awaiting acknowledgment.

Parameters

- **queue** (*str*) – The queue name
- **nowait** (*bool*) – Do not send a reply method

queue_unbind(*queue=''*, *exchange=''*, *routing_key=''*, *arguments=None*)
Unbind a queue from an exchange

This method unbinds a queue from an exchange.

Parameters

- **queue** (*str*) – The queue name
- **exchange** (*str*) – The exchange name
- **routing_key** (*str*) – Routing key of binding
- **arguments** (*dict*) – Arguments of binding

tx_commit()
Commit the current transaction

This method commits all message publications and acknowledgments performed in the current transaction. A new transaction starts immediately after a commit.

tx_rollback()
Abandon the current transaction

This method abandons all message publications and acknowledgments performed in the current transaction. A new transaction starts immediately after a rollback. Note that unacked messages will not be automatically redelivered by rollback; if that is required an explicit recover call should be issued.

tx_select()
Select standard transaction mode

This method sets the channel to use standard transactions. The client must use this method at least once on a channel before using the Commit or Rollback methods.

2.3 Channel

A `Channel` is created on an active connection using the *`Connection.channel()`* method. Channels can act as normal Python objects:

```
conn = rabbitpy.Connection()
chan = conn.channel()
chan.enable_publisher_confirms()
chan.close()
```

or as a Python context manager (See **PEP 0343**):

```
with rabbitpy.Connection() as conn:
    with conn.channel() as chan:
        chan.enable_publisher_confirms()
```

When they are used as a context manager with the *with* statement, when your code exits the block, the channel will automatically close, issuing a clean shutdown with RabbitMQ via the `Channel.Close` RPC request.

You should be aware that if you perform actions on a channel with exchanges, queues, messages or transactions that RabbitMQ does not like, it will close the channel by sending an AMQP `Channel.Close` RPC request to your application. Upon receipt of such a request, rabbitpy will raise the appropriate exception referenced in the request.

2.3.1 API Documentation

class rabbitpy.**Channel**(*channel_id*, *server_capabilities*, *events*, *exception_queue*, *read_queue*, *write_queue*, *maximum_frame_size*, *write_trigger*, *blocking_read=False*)

The Channel object is the communications object used by Exchanges, Messages, Queues, and Transactions. It is created by invoking the `rabbitpy.Connection.channel()` method. It can act as a context manager, allowing for quick shorthand use:

```
with connection.channel():
   # Do something
```

To create a new channel, invoke py:meth:*rabbitpy.connection.Connection.channel*

To improve performance, pass blocking_read to True. Note that doing so prevents `KeyboardInterrupt`/CTRL-C from exiting the Python interpreter.

Parameters

- **channel_id** (*int*) – The channel # to use for this instance
- **server_capabilities** (*dict*) – Features the server supports
- **rabbitpy.Events** (*events*) – Event management object
- **exception_queue** (*queue.Queue*) – Exception queue
- **read_queue** (*queue.Queue*) – Queue to read pending frames from
- **write_queue** (*queue.Queue*) – Queue to write pending AMQP objs to
- **maximum_frame_size** (*int*) – The max frame size for msg bodies
- **write_trigger** (*socket*) – Write to this socket to break IO waiting
- **blocking_read** (*bool*) – Use blocking Queue.get to improve performance

Raises rabbitpy.exceptions.RemoteClosedChannelException

Raises rabbitpy.exceptions.AMQPException

close()

Close the channel, cancelling any active consumers, purging the read queue, while looking to see if a Basic.Nack should be sent, sending it if so.

enable_publisher_confirms()

Turn on Publisher Confirms. If confirms are turned on, the Message.publish command will return a bool indicating if a message has been successfully published.

id

Return the channel id

Return type int

open()

Open the channel, invoked directly upon creation by the Connection

prefetch_count(*value*, *all_channels=False*)

Set a prefetch count for the channel (or all channels on the same connection).

Parameters

- **value** (*int*) – The prefetch count to set
- **all_channels** (*bool*) – Set the prefetch count on all channels on the same connection

prefetch_size(*value*, *all_channels=False*)
Set a prefetch size in bytes for the channel (or all channels on the same connection).

Parameters

- **value** (*int*) – The prefetch size to set
- **all_channels** (*bool*) – Set the prefetch size on all channels on the same connection

publisher_confirms
Returns True if publisher confirms are enabled.

Return type bool

recover(*requeue=False*)
Recover all unacknowledged messages that are associated with this channel.

Parameters **requeue** (*bool*) – Requeue the message

2.4 Connection

rabbitpy Connection objects are used to connect to RabbitMQ. They provide a thread-safe connection to RabbitMQ that is used to authenticate and send all channel based RPC commands over. Connections use AMQP URI syntax for specifying the all of the connection information, including any connection negotiation options, such as the heartbeat interval. For more information on the various query parameters that can be specified, see the official documentation.

A `Connection` is a normal python object that you use:

```
conn = rabbitpy.Connection('amqp://guest:guest@localhost:5672/%2F')
conn.close()
```

or it can be used as a Python context manager (See **PEP 0343**):

```
with rabbitpy.Connection() as conn:
    # Foo
```

When it is used as a context manager with the *with* statement, when your code exits the block, the connection will automatically close.

If RabbitMQ remotely closes your connection via the AMQP *Connection.Close* RPC request, rabbitpy will raise the appropriate exception referenced in the request.

If heartbeats are enabled (default: 5 minutes) and RabbitMQ does not send a heartbeat request in >= 2 heartbeat intervals, a *ConnectionResetException* will be raised.

2.4.1 API Documentation

class rabbitpy.**Connection**(*url=None*)
The Connection object is responsible for negotiating a connection and managing its state. When creating a new instance of the Connection object, if no URL is passed in, it uses the default connection parameters of localhost port 5672, virtual host / with the guest/guest username/password combination. Represented as a AMQP URL the connection information is:

```
amqp://guest:guest@localhost:5672/%2F
```

To use a different connection, pass in a AMQP URL that follows the standard format:

```
[scheme]://[username]:[password]@[host]:[port]/[virtual_host]
```

The following example connects to the test virtual host on a RabbitMQ server running at 192.168.1.200 port 5672 as the user "www" and the password rabbitmq:

```
amqp://admin192.168.1.200:5672/test
```

Note: You should be aware that most connection exceptions may be raised during the use of all functionality in the library.

Parameters **url** (*str*) – The AMQP connection URL

Raises rabbitpy.exceptions.AMQPException

Raises rabbitpy.exceptions.ConnectionException

Raises rabbitpy.exceptions.ConnectionResetException

Raises rabbitpy.exceptions.RemoteClosedException

blocked
Indicates if the connection is blocked from publishing by RabbitMQ.

This flag indicates communication from RabbitMQ that the connection is blocked using the Connection.Blocked RPC notification from RabbitMQ that was added in RabbitMQ 3.2.

@TODO If RabbitMQ version < 3.2, use the HTTP management API to query the value

Return type bool

capabilities
Return the RabbitMQ Server capabilities from the connection negotiation process.

Return type dict

channel(*blocking_read=False*)
Create a new channel

If blocking_read is True, the cross-thread Queue.get use will use blocking operations that lower resource utilization and increase throughput. However, due to how Python's blocking Queue.get is implemented, KeyboardInterrupt is not raised when CTRL-C is pressed.

Parameters **blocking_read** (*bool*) – Enable for higher throughput

Raises rabbitpy.exceptions.AMQPException

Raises rabbitpy.exceptions.RemoteClosedChannelException

close()
Close the connection, including all open channels

server_properties
Return the RabbitMQ Server properties from the connection negotiation process.

Return type dict

2.5 Exceptions

rabbitpy contains two types of exceptions, exceptions that are specific to rabbitpy and exceptions that are raises as the result of a Channel or Connection closure from RabbitMQ. These exceptions will be raised to let you know when you have performed an action like redeclared a pre-existing queue with different values. Consider the following example:

```
>>> import rabbitpy
>>>
>>> with rabbitpy.Connection() as connection:
...     with connection.channel() as channel:
...         queue = rabbitpy.Queue(channel, 'exception-test')
...         queue.durable = True
...         queue.declare()
...         queue.durable = False
...         queue.declare()
...
Traceback (most recent call last):
  File "<stdin>", line 7, in <module>
  File "rabbitpy/connection.py", line 131, in __exit__
    self._shutdown_connection()
  File "rabbitpy/connection.py", line 469, in _shutdown_connection
    self._channels[chan_id].close()
  File "rabbitpy/channel.py", line 124, in close
    super(Channel, self).close()
  File "rabbitpy/base.py", line 185, in close
    self.rpc(frame_value)
  File "rabbitpy/base.py", line 199, in rpc
    self._write_frame(frame_value)
  File "rabbitpy/base.py", line 311, in _write_frame
    raise exception
rabbitpy.exceptions.AMQPPreconditionFailed: <pamqp.specification.Channel.Close object at 0x10e86bd50>
```

In this example, the channel that was created on the second line was closed and RabbitMQ is raising the *AMQPPreconditionFailed* exception via RPC sent to your application using the AMQP Channel.Close method.

2.5.1 Exceptions that may be raised by rabbitpy during use

exception rabbitpy.exceptions.**AMQPAccessRefused**
The client attempted to work with a server entity to which it has no access due to security settings.

exception rabbitpy.exceptions.**AMQPChannelError**
The client attempted to work with a channel that had not been correctly opened. This most likely indicates a fault in the client layer.

exception rabbitpy.exceptions.**AMQPCommandInvalid**
The client sent an invalid sequence of frames, attempting to perform an operation that was considered invalid by the server. This usually implies a programming error in the client.

exception rabbitpy.exceptions.**AMQPConnectionForced**
An operator intervened to close the connection for some reason. The client may retry at some later date.

exception rabbitpy.exceptions.**AMQPContentTooLarge**
The client attempted to transfer content larger than the server could accept at the present time. The client may retry at a later time.

exception rabbitpy.exceptions.**AMQPException**
Base exception of all AMQP exceptions.

exception rabbitpy.exceptions.**AMQPFrameError**
The sender sent a malformed frame that the recipient could not decode. This strongly implies a programming error in the sending peer.

exception rabbitpy.exceptions.**AMQPInternalError**
The server could not complete the method because of an internal error. The server may require intervention by an operator in order to resume normal operations.

exception rabbitpy.exceptions.**AMQPInvalidPath**
The client tried to work with an unknown virtual host.

exception rabbitpy.exceptions.**AMQPNoConsumers**
When the exchange cannot deliver to a consumer when the immediate flag is set. As a result of pending data on the queue or the absence of any consumers of the queue.

exception rabbitpy.exceptions.**AMQPNoRoute**
Undocumented AMQP Soft Error

exception rabbitpy.exceptions.**AMQPNotAllowed**
The client tried to work with some entity in a manner that is prohibited by the server, due to security settings or by some other criteria.

exception rabbitpy.exceptions.**AMQPNotFound**
The client attempted to work with a server entity that does not exist.

exception rabbitpy.exceptions.**AMQPNotImplemented**
The client tried to use functionality that is not implemented in the server.

exception rabbitpy.exceptions.**AMQPPreconditionFailed**
The client requested a method that was not allowed because some precondition failed.

exception rabbitpy.exceptions.**AMQPResourceError**
The server could not complete the method because it lacked sufficient resources. This may be due to the client creating too many of some type of entity.

exception rabbitpy.exceptions.**AMQPResourceLocked**
The client attempted to work with a server entity to which it has no access because another client is working with it.

exception rabbitpy.exceptions.**AMQPSyntaxError**
The sender sent a frame that contained illegal values for one or more fields. This strongly implies a programming error in the sending peer.

exception rabbitpy.exceptions.**AMQPUnexpectedFrame**
The peer sent a frame that was not expected, usually in the context of a content header and body. This strongly indicates a fault in the peer's content processing.

exception rabbitpy.exceptions.**ActionException**
Raised when an action is taken on a Rabbitpy object that is not supported due to the state of the object. An example would be trying to ack a Message object when the message object was locally created and not sent by RabbitMQ via an AMQP Basic.Get or Basic.Consume.

exception rabbitpy.exceptions.**ChannelClosedException**
Raised when an action is attempted on a channel that is closed.

exception rabbitpy.exceptions.**ConnectionException**
Raised when Rabbitpy can not connect to the specified server and if a connection fails and the RabbitMQ version does not support the authentication_failure_close feature added in RabbitMQ 3.2.

exception rabbitpy.exceptions.**ConnectionResetException**
Raised if the socket level connection was reset. This can happen due to the loss of network connection or socket timeout, or more than 2 missed heartbeat intervals if heartbeats are enabled.

exception rabbitpy.exceptions.**MessageReturnedException**
Raised if the RabbitMQ sends a message back to a publisher via the Basic.Return RPC call.

exception rabbitpy.exceptions.**NoActiveTransactionError**
Raised when a transaction method is issued but the transaction has not been initiated.

exception rabbitpy.exceptions.**NotConsumingError**
Raised Queue.cancel_consumer() is invoked but the queue is not actively consuming.

exception rabbitpy.exceptions.**NotSupportedError**
Raised when a feature is requested that is not supported by the RabbitMQ server.

exception rabbitpy.exceptions.**RabbitpyException**
Base exception of all rabbitpy exceptions.

exception rabbitpy.exceptions.**RemoteCancellationException**
Raised if RabbitMQ cancels an active consumer

exception rabbitpy.exceptions.**RemoteClosedChannelException**
Raised if RabbitMQ closes the channel and the reply_code in the Channel.Close RPC request does not have a mapped exception in Rabbitpy.

exception rabbitpy.exceptions.**RemoteClosedException**
Raised if RabbitMQ closes the connection and the reply_code in the Connection.Close RPC request does not have a mapped exception in Rabbitpy.

exception rabbitpy.exceptions.**TooManyChannelsError**
Raised if an application attempts to create a channel, exceeding the maximum number of channels (MAXINT or 2,147,483,647) available for a single connection. Note that each time a channel object is created, it will take a new channel id. If you create and destroy 2,147,483,648 channels, this exception will be raised.

exception rabbitpy.exceptions.**UnexpectedResponseError**
Raised when an RPC call is made to RabbitMQ but the response it sent back is not recognized.

2.6 Exchange

The *Exchange* class is used to work with RabbitMQ exchanges on an open channel. The following example shows how you can create an exchange using the *rabbitpy.Exchange* class.

```
import rabbitpy

with rabbitpy.Connection() as connection:
    with connection.channel() as channel:
        exchange = rabbitpy.Exchange(channel, 'my-exchange')
        exchange.declare()
```

In addition, there are four convenience classes (*DirectExchange*, *FanoutExchange*, *HeadersExchange*, and *TopicExchange*) for creating each built-in exchange type in RabbitMQ.

2.6.1 API Documentation

class rabbitpy.**Exchange**(*channel*, *name*, *exchange_type='direct'*, *durable=False*, *auto_delete=False*, *arguments=None*)
Exchange class for interacting with an exchange in RabbitMQ including declaration, binding and deletion.

Parameters

- **channel** (rabbitpy.channel.Channel) – The channel object to communicate on
- **name** (*str*) – The name of the exchange
- **exchange_type** (*str*) – The exchange type

- **durable** (*bool*) – Request a durable exchange
- **auto_delete** (*bool*) – Automatically delete when not in use
- **arguments** (*dict*) – Optional key/value arguments

bind(*source*, *routing_key=None*)
Bind to another exchange with the routing key.

Parameters

- **source** (str or *rabbitpy.Exchange*) – The exchange to bind to
- **routing_key** (*str*) – The routing key to use

declare(*passive=False*)
Declare the exchange with RabbitMQ. If passive is True and the command arguments do not match, the channel will be closed.

Parameters **passive** (*bool*) – Do not actually create the exchange

delete(*if_unused=False*)
Delete the exchange from RabbitMQ.

Parameters **if_unused** (*bool*) – Delete only if unused

unbind(*source*, *routing_key=None*)
Unbind the exchange from the source exchange with the routing key. If routing key is None, use the queue or exchange name.

Parameters

- **source** (str or *rabbitpy.Exchange*) – The exchange to unbind from
- **routing_key** (*str*) – The routing key that binds them

class rabbitpy.**DirectExchange**(*channel*, *name*, *durable=False*, *auto_delete=False*, *arguments=None*)
The DirectExchange class is used for interacting with direct exchanges only.

Parameters

- **channel** (rabbitpy.channel.Channel) – The channel object to communicate on
- **name** (*str*) – The name of the exchange
- **durable** (*bool*) – Request a durable exchange
- **auto_delete** (*bool*) – Automatically delete when not in use
- **arguments** (*dict*) – Optional key/value arguments

bind(*source*, *routing_key=None*)
Bind to another exchange with the routing key.

Parameters

- **source** (str or *rabbitpy.Exchange*) – The exchange to bind to
- **routing_key** (*str*) – The routing key to use

declare(*passive=False*)
Declare the exchange with RabbitMQ. If passive is True and the command arguments do not match, the channel will be closed.

Parameters **passive** (*bool*) – Do not actually create the exchange

delete(*if_unused=False*)
Delete the exchange from RabbitMQ.

Parameters **if_unused** (*bool*) – Delete only if unused

unbind(*source*, *routing_key=None*)
Unbind the exchange from the source exchange with the routing key. If routing key is None, use the queue or exchange name.

Parameters

- **source** (str or *rabbitpy.Exchange*) – The exchange to unbind from
- **routing_key** (*str*) – The routing key that binds them

class rabbitpy.**FanoutExchange**(*channel*, *name*, *durable=False*, *auto_delete=False*, *arguments=None*)
The FanoutExchange class is used for interacting with fanout exchanges only.

Parameters

- **channel** (rabbitpy.channel.Channel) – The channel object to communicate on
- **name** (*str*) – The name of the exchange
- **durable** (*bool*) – Request a durable exchange
- **auto_delete** (*bool*) – Automatically delete when not in use
- **arguments** (*dict*) – Optional key/value arguments

bind(*source*, *routing_key=None*)
Bind to another exchange with the routing key.

Parameters

- **source** (str or *rabbitpy.Exchange*) – The exchange to bind to
- **routing_key** (*str*) – The routing key to use

declare(*passive=False*)
Declare the exchange with RabbitMQ. If passive is True and the command arguments do not match, the channel will be closed.

Parameters **passive** (*bool*) – Do not actually create the exchange

delete(*if_unused=False*)
Delete the exchange from RabbitMQ.

Parameters **if_unused** (*bool*) – Delete only if unused

unbind(*source*, *routing_key=None*)
Unbind the exchange from the source exchange with the routing key. If routing key is None, use the queue or exchange name.

Parameters

- **source** (str or *rabbitpy.Exchange*) – The exchange to unbind from
- **routing_key** (*str*) – The routing key that binds them

class rabbitpy.**HeadersExchange**(*channel*, *name*, *durable=False*, *auto_delete=False*, *arguments=None*)
The HeadersExchange class is used for interacting with direct exchanges only.

Parameters

- **channel** (rabbitpy.channel.Channel) – The channel object to communicate on

- **name** (*str*) – The name of the exchange
- **durable** (*bool*) – Request a durable exchange
- **auto_delete** (*bool*) – Automatically delete when not in use
- **arguments** (*dict*) – Optional key/value arguments

bind(*source*, *routing_key=None*)
Bind to another exchange with the routing key.

Parameters

- **source** (str or *rabbitpy.Exchange*) – The exchange to bind to
- **routing_key** (*str*) – The routing key to use

declare(*passive=False*)
Declare the exchange with RabbitMQ. If passive is True and the command arguments do not match, the channel will be closed.

Parameters **passive** (*bool*) – Do not actually create the exchange

delete(*if_unused=False*)
Delete the exchange from RabbitMQ.

Parameters **if_unused** (*bool*) – Delete only if unused

unbind(*source*, *routing_key=None*)
Unbind the exchange from the source exchange with the routing key. If routing key is None, use the queue or exchange name.

Parameters

- **source** (str or *rabbitpy.Exchange*) – The exchange to unbind from
- **routing_key** (*str*) – The routing key that binds them

class rabbitpy.**TopicExchange**(*channel*, *name*, *durable=False*, *auto_delete=False*, *arguments=None*)
The TopicExchange class is used for interacting with topic exchanges only.

Parameters

- **channel** (rabbitpy.channel.Channel) – The channel object to communicate on
- **name** (*str*) – The name of the exchange
- **durable** (*bool*) – Request a durable exchange
- **auto_delete** (*bool*) – Automatically delete when not in use
- **arguments** (*dict*) – Optional key/value arguments

bind(*source*, *routing_key=None*)
Bind to another exchange with the routing key.

Parameters

- **source** (str or *rabbitpy.Exchange*) – The exchange to bind to
- **routing_key** (*str*) – The routing key to use

declare(*passive=False*)
Declare the exchange with RabbitMQ. If passive is True and the command arguments do not match, the channel will be closed.

Parameters **passive** (*bool*) – Do not actually create the exchange

delete(*if_unused=False*)
Delete the exchange from RabbitMQ.

Parameters **if_unused** (*bool*) – Delete only if unused

unbind(*source*, *routing_key=None*)
Unbind the exchange from the source exchange with the routing key. If routing key is None, use the queue or exchange name.

Parameters

- **source** (str or *rabbitpy.Exchange*) – The exchange to unbind from
- **routing_key** (*str*) – The routing key that binds them

2.7 Message

The `Message` class is used to create messages that you intend to publish to RabbitMQ and is created when a message is received by RabbitMQ by a consumer or as the result of a *Queue.get()* request.

2.7.1 API Documentation

class `rabbitpy.`**Message**(*channel*, *body_value*, *properties=None*, *auto_id=False*, *opinionated=False*)
Created by both rabbitpy internally when a message is delivered or returned from RabbitMQ and by implementing applications, the Message class is used to publish a message to and access and respond to a message from RabbitMQ.

When specifying properties for a message, pass in a dict of key value items that match the AMQP Basic.Properties specification with a small caveat.

Due to an overlap in the AMQP specification and the Python keyword `type`, the `type` property is referred to as `message_type`.

The following is a list of the available properties:

•app_id

•content_type

•content_encoding

•correlation_id

•delivery_node

•expiration

•headers

•message_id

•message_type

•priority

•reply_to

•timestamp

•user_id

Automated features

When passing in the body value, if it is a dict or list, it will automatically be JSON serialized and the content type `application/json` will be set on the message properties.

When publishing a message to RabbitMQ, if the opinionated value is `True` and no `message_id` value was passed in as a property, a UUID will be generated and specified as a property of the message.

Additionally, if opinionated is `True` and the `timestamp` property is not specified when passing in `properties`, the current Unix epoch value will be set in the message properties.

Note: As of 0.21.0 `auto_id` is deprecated in favor of

`opinionated` and it will be removed in a future version. As of 0.22.0 `opinionated` is defaulted to `False`.

Parameters

- **channel** (`rabbitpy.channel.Channel`) – The channel object for the message object to act upon
- **or dict or list body_value** (*str*) – The message body
- **properties** (*dict*) – A dictionary of message properties
- **auto_id** (*bool*) – Add a message id if no properties were passed in.
- **opinionated** (*bool*) – Automatically populate properties if True

Raises **KeyError** – Raised when an invalid property is passed in

ack(*all_previous=False*)
Acknowledge receipt of the message to RabbitMQ. Will raise an ActionException if the message was not received from a broker.

Raises ActionException

delivery_tag
Return the delivery tag for a message that was delivered or gotten from RabbitMQ.

Return type int or None

exchange
Return the source exchange for a message that was delivered or gotten from RabbitMQ.

Return type string or None

json()
Deserialize the message body if it is JSON, returning the value.

Return type any

nack(*requeue=False*, *all_previous=False*)
Negatively acknowledge receipt of the message to RabbitMQ. Will raise an ActionException if the message was not received from a broker.

Parameters

- **requeue** (*bool*) – Requeue the message
- **all_previous** (*bool*) – Nack all previous unacked messages up to and including this one

Raises ActionException

pprint (*properties=False*)
Print a formatted representation of the message.

Parameters **properties** (*bool*) – Include properties in the representation

publish (*exchange*, *routing_key=''*, *mandatory=False*, *immediate=False*)
Publish the message to the exchange with the specified routing key.

In Python 2 if the message is a `unicode` value it will be converted to a `str` using `str.encode('UTF-8')`. If you do not want the auto-conversion to take place, set the body to a `str` or `bytes` value prior to publishing.

In Python 3 if the message is a `str` value it will be converted to a `bytes` value using `bytes(value.encode('UTF-8'))`. If you do not want the auto-conversion to take place, set the body to a `bytes` value prior to publishing.

Parameters

- **exchange** (str or *rabbitpy.Exchange*) – The exchange to publish the message to
- **routing_key** (*str*) – The routing key to use
- **mandatory** (*bool*) – Requires the message is published
- **immediate** (*bool*) – Request immediate delivery

Returns bool or None

Raises rabbitpy.exceptions.MessageReturnedException

redelivered
Indicates if this message may have been delivered before (but not acknowledged)"

Return type bool or None

reject (*requeue=False*)
Reject receipt of the message to RabbitMQ. Will raise an ActionException if the message was not received from a broker.

Parameters **requeue** (*bool*) – Requeue the message

Raises ActionException

routing_key
Return the routing_key for a message that was delivered or gotten from RabbitMQ.

Return type int or None

2.8 Queue

The *Queue* class is used to work with RabbitMQ queues on an open channel. The following example shows how you can create a queue using the *Queue.declare* method.

```
import rabbitpy

with rabbitpy.Connection() as connection:
    with connection.channel() as channel:
        queue = rabbitpy.Queue(channel, 'my-queue')
        queue.durable = True
        queue.declare()
```

To consume messages you can iterate over the Queue object itself if the defaults for the `Queue.__iter__()` method work for your needs:

```
with conn.channel() as channel:
    for message in rabbitpy.Queue(channel, 'example'):
        print 'Message: %r' % message
        message.ack()
```

or by the *Queue.consume()* method if you would like to specify *no_ack*, *prefetch_count*, or *priority*:

```
with conn.channel() as channel:
    queue = rabbitpy.Queue(channel, 'example')
    for message in queue.consume():
        print 'Message: %r' % message
        message.ack()
```

Warning: If you use either the `Queue` as an iterator method or `Queue.consume()` method of consuming messages in PyPy, you must manually invoke `Queue.stop_consuming()`. This is due to PyPy not predictably cleaning up after the generator used for allowing the iteration over messages. Should your code want to test to see if the code is being executed in PyPy, you can evaluate the boolean `rabbitpy.PYPY` constant value.

2.8.1 API Documentation

class `rabbitpy.`**Queue**(*channel*, *name=''*, *durable=False*, *exclusive=False*, *auto_delete=False*, *max_length=None*, *message_ttl=None*, *expires=None*, *dead_letter_exchange=None*, *dead_letter_routing_key=None*, *arguments=None*)

Create and manage RabbitMQ queues.

Parameters

- **channel** (`rabbitpy.channel.Channel`) – The channel object to communicate on
- **name** (*str*) – The name of the queue
- **exclusive** (*bool*) – Queue can only be used by this channel and will auto-delete once the channel is closed.
- **durable** (*bool*) – Indicates if the queue should survive a RabbitMQ is restart
- **auto_delete** (*bool*) – Automatically delete when all consumers disconnect
- **max_length** (*int*) – Maximum queue length
- **message_ttl** (*int*) – Time-to-live of a message in milliseconds
- **expires** (*int*) – Milliseconds until a queue is removed after becoming idle
- **dead_letter_exchange** (*str*) – Dead letter exchange for rejected messages
- **dead_letter_routing_key** (*str*) – Routing key for dead lettered messages
- **arguments** (*dict*) – Custom arguments for the queue

Raises rabbitpy.exceptions.RemoteClosedChannelException

Raises rabbitpy.exceptions.RemoteCancellationException

__init__(*channel*, *name=''*, *durable=False*, *exclusive=False*, *auto_delete=False*, *max_length=None*, *message_ttl=None*, *expires=None*, *dead_letter_exchange=None*, *dead_letter_routing_key=None*, *arguments=None*)

Create a new Queue object instance. Only the *rabbitpy.Channel* object is required.

__iter__()
Quick way to consume messages using defaults of no_ack=False, prefetch and priority not set.

Yields rabbitpy.message.Message

__len__()
Return the pending number of messages in the queue by doing a passive Queue declare.

Return type int

__setattr__(*name*, *value*)
Validate the data types for specific attributes when setting them, otherwise fall throw to the parent __setattr__

Parameters

- **name** (*str*) – The attribute to set
- **value** (*mixed*) – The value to set

Raises ValueError

bind(*source*, *routing_key=None*, *arguments=None*)
Bind the queue to the specified exchange or routing key.

Parameters

- **source** (str or `rabbitpy.exchange.Exchange` exchange) – The exchange to bind to
- **routing_key** (*str*) – The routing key to use
- **arguments** (*dict*) – Optional arguments for for RabbitMQ

Returns bool

consume(*no_ack=False*, *prefetch=None*, *priority=None*)
Consume messages from the queue as a `generator`:

You can use this method instead of the queue object as an iterator if you need to alter the prefect count, set the consumer priority or consume in no_ack mode.

New in version 0.26.

Parameters

- **no_ack** (*bool*) – Do not require acknowledgements
- **prefetch** (*int*) – Set a prefetch count for the channel
- **priority** (*int*) – Consumer priority

Return type `generator`

Raises rabbitpy.exceptions.RemoteCancellationException

consume_messages(*no_ack=False*, *prefetch=None*, *priority=None*)
Consume messages from the queue as a generator.

> **Warning:** This method is deprecated in favor of `Queue.consume()` and will be removed in future releases.

Deprecated since version 0.26.

You can use this message instead of the queue object as an iterator if you need to alter the prefect count, set the consumer priority or consume in no_ack mode.

Parameters

- **no_ack** (*bool*) – Do not require acknowledgements
- **prefetch** (*int*) – Set a prefetch count for the channel
- **priority** (*int*) – Consumer priority

Return type Generator

Raises rabbitpy.exceptions.RemoteCancellationException

consumer(*no_ack=False*, *prefetch=None*, *priority=None*)
Method for returning the contextmanager for consuming messages. You should not use this directly.

> **Warning:** This method is deprecated and will be removed in a future release.

Deprecated since version 0.26.

Parameters

- **no_ack** (*bool*) – Do not require acknowledgements
- **prefetch** (*int*) – Set a prefetch count for the channel
- **priority** (*int*) – Consumer priority

Returns None

declare(*passive=False*)
Declare the queue on the RabbitMQ channel passed into the constructor, returning the current message count for the queue and its consumer count as a tuple.

Parameters **passive** (*bool*) – Passive declare to retrieve message count and consumer count information

Returns Message count, Consumer count

Return type tuple(int, int)

delete(*if_unused=False*, *if_empty=False*)
Delete the queue

Parameters

- **if_unused** (*bool*) – Delete only if unused
- **if_empty** (*bool*) – Delete only if empty

get(*acknowledge=True*)
Request a single message from RabbitMQ using the Basic.Get AMQP command.

Parameters **acknowledge** (*bool*) – Let RabbitMQ know if you will manually acknowledge or negatively acknowledge the message after each get.

Return type rabbitpy.message.Message or None

ha_declare(*nodes=None*)
Declare a the queue as highly available, passing in a list of nodes the queue should live on. If no nodes are passed, the queue will be declared across all nodes in the cluster.

Parameters **nodes** (*list*) – A list of nodes to declare. If left empty, queue will be declared on all cluster nodes.

Returns Message count, Consumer count

Return type tuple(int, int)

purge()
Purge the queue of all of its messages.

stop_consuming()
Stop consuming messages. This is usually invoked if you want to cancel your consumer from outside the context manager or generator.

If you invoke this, there is a possibility that the generator method will return None instead of a *rabbitpy.Message*.

unbind(*source*, *routing_key=None*)
Unbind queue from the specified exchange where it is bound the routing key. If routing key is None, use the queue name.

Parameters

- **source** (str or rabbitpy.exchange.Exchange exchange) – The exchange to unbind from
- **routing_key** (*str*) – The routing key that binds them

2.9 Transactions

The *Tx* or transaction class implements transactional functionality with RabbitMQ and allows for any AMQP command to be issued, then committed or rolled back.

It can be used as a normal Python object:

```
with rabbitpy.Connection() as connection:
    with connection.channel() as channel:
        tx = rabbitpy.Tx(channel)
        tx.select()
        exchange = rabbitpy.Exchange(channel, 'my-exchange')
        exchange.declare()
        tx.commit()
```

Or as a context manager (See **PEP 0343**) where the transaction will automatically be started and committed for you:

```
with rabbitpy.Connection() as connection:
    with connection.channel() as channel:
        with rabbitpy.Tx(channel) as tx:
            exchange = rabbitpy.Exchange(channel, 'my-exchange')
            exchange.declare()
```

In the event of an exception exiting the block when used as a context manager, the transaction will be rolled back for you automatically.

2.9.1 API Documentation

class rabbitpy.**Tx**(*channel*)
Work with transactions

The Tx class allows publish and ack operations to be batched into atomic units of work. The intention is that all publish and ack requests issued within a transaction will complete successfully or none of them will. Servers SHOULD implement atomic transactions at least where all publish or ack requests affect a single queue. Transactions that cover multiple queues may be non-atomic, given that queues can be created and destroyed

asynchronously, and such events do not form part of any transaction. Further, the behaviour of transactions with respect to the immediate and mandatory flags on Basic.Publish methods is not defined.

Parameters **channel** (rabbitpy.channel.Channel) – The channel object to start the transaction on

commit()
Commit the current transaction

This method commits all message publications and acknowledgments performed in the current transaction. A new transaction starts immediately after a commit.

Raises rabbitpy.exceptions.NoActiveTransactionError

Return type bool

rollback()
Abandon the current transaction

This method abandons all message publications and acknowledgments performed in the current transaction. A new transaction starts immediately after a rollback. Note that unacked messages will not be automatically redelivered by rollback; if that is required an explicit recover call should be issued.

Raises rabbitpy.exceptions.NoActiveTransactionError

Return type bool

select()
Select standard transaction mode

This method sets the channel to use standard transactions. The client must use this method at least once on a channel before using the Commit or Rollback methods.

Return type bool

CHAPTER 3

Examples

3.1 Message Consumer

The following example will subscribe to a queue named "example" and consume messages until CTRL-C is pressed:

```
import rabbitpy

with rabbitpy.Connection('amqp://guest:guest@localhost:5672/%2f') as conn:
    with conn.channel() as channel:
        queue = rabbitpy.Queue(channel, 'example')

        # Exit on CTRL-C
        try:
            # Consume the message
            for message in queue:
                message.pprint(True)
                message.ack()

        except KeyboardInterrupt:
            print 'Exited consumer'
```

3.2 Message Getter

The following example will get a single message at a time from the "example" queue as long as there are messages in it. It uses `len(queue)` to check the current queue depth while it is looping:

```
import rabbitpy

with rabbitpy.Connection('amqp://guest:guest@localhost:5672/%2f') as conn:
    with conn.channel() as channel:
        queue = rabbitpy.Queue(channel, 'example')

        while len(queue) > 0:
            message = queue.get()
            print 'Message:'
            print ' ID: %s' % message.properties['message_id']
            print ' Time: %s' % message.properties['timestamp'].isoformat()
            print ' Body: %s' % message.body
            message.ack()
```

```
            print 'There are %i more messages in the queue' % len(queue)
```

3.3 Declaring HA Queues

The following example will create a HA `queue` on each node in a RabbitMQ cluster.:

```
import rabbitpy

with rabbitpy.Connection('amqp://guest:guest@localhost:5672/%2f') as conn:
    with conn.channel() as channel:
        queue = rabbitpy.Queue(channel, 'example')
        queue.ha_declare()
```

3.4 Mandatory Publishing

The following example uses RabbitMQ's Publisher Confirms feature to allow for validation that the message was successfully published:

```
import rabbitpy

# Connect to RabbitMQ on localhost, port 5672 as guest/guest
with rabbitpy.Connection('amqp://guest:guest@localhost:5672/%2f') as conn:

    # Open the channel to communicate with RabbitMQ
    with conn.channel() as channel:

        # Turn on publisher confirmations
        channel.enable_publisher_confirms()

        # Create the message to publish
        message = rabbitpy.Message(channel, 'message body value')

        # Publish the message, looking for the return value to be a bool True/False
        if message.publish('test_exchange', 'test-routing-key', mandatory=True):
            print 'Message publish confirmed by RabbitMQ'
        else:
            print 'RabbitMQ indicates message publishing failure'
```

3.5 Transactional Publisher

The following example uses RabbitMQ's Transactions feature to send the message, then roll it back:

```
import rabbitpy

# Connect to RabbitMQ on localhost, port 5672 as guest/guest
with rabbitpy.Connection('amqp://guest:guest@localhost:5672/%2f') as conn:

    # Open the channel to communicate with RabbitMQ
    with conn.channel() as channel:

        # Start the transaction
```

```
        tx = rabbitpy.Tx(channel)
        tx.select()

        # Create the message to publish & publish it
        message = rabbitpy.Message(channel, 'message body value')
        message.publish('test_exchange', 'test-routing-key')

        # Rollback the transaction
        tx.rollback()
```

CHAPTER 4

Issues

Please report any issues to the Github repo at https://github.com/gmr/rabbitpy/issues

CHAPTER 5

Source

rabbitpy source is available on Github at https://github.com/gmr/rabbitpy

CHAPTER 6

Version History

See `history`

CHAPTER 7

Inspiration

rabbitpy's simple and more pythonic interface is inspired by Kenneth Reitz's awesome work on requests.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

r

Symbols

A

B

C

D