discord.py Documentation

Release 0.16.12

Rapptz

Jan 06, 2018

Contents

Contents:

New in version 0.6.0.

CHAPTER 1

Setting Up Logging

discord.py logs errors and debug information via the logging python module. It is strongly recommended that the logging module is configured, as no errors or warnings will be output if it is not set up. Configuration of the `logging` module can be as simple as:

```
import logging

logging.basicConfig(level=logging.INFO)
```

Placed at the start of the application. This will output the logs from discord as well as other libraries that uses the `logging` module directly to the console.

The optional `level` argument specifies what level of events to log out and can any of `CRITICAL`, `ERROR`, `WARNING`, `INFO`, and `DEBUG` and if not specified defaults to `WARNING`.

More advanced setups are possible with the `logging` module. To for example write the logs to a file called `discord.log` instead of outputting them to to the console the following snippet can be used:

```
import discord
import logging

logger = logging.getLogger('discord')
logger.setLevel(logging.DEBUG)
handler = logging.FileHandler(filename='discord.log', encoding='utf-8', mode='w')
handler.setFormatter(logging.Formatter('%(asctime)s:%(levelname)s:%(name)s:
↪%(message)s'))
logger.addHandler(handler)
```

This is recommended, especially at verbose levels such as `INFO`, and `DEBUG` as there are a lot of events logged and it would clog the stdout of your program.

For more information, check the documentation and tutorial of the logging module.

CHAPTER 2

What's New

This page keeps a detailed human friendly rendering of what's new and changed in specific versions.

2.1 v0.16.6

2.1.1 Bug Fixes

- Fix issue with *Client.create_server()* that made it stop working.
- Fix main thread being blocked upon calling `StreamPlayer.stop`.
- Handle HEARTBEAT_ACK and resume gracefully when it occurs.
- Fix race condition when pre-emptively rate limiting that caused releasing an already released lock.
- Fix invalid state errors when immediately cancelling a coroutine.

2.2 v0.16.1

This release is just a bug fix release with some better rate limit implementation.

2.2.1 Bug Fixes

- Servers are now properly chunked for user bots.
- The CDN URL is now used instead of the API URL for assets.
- Rate limit implementation now tries to use header information if possible.
- Event loop is now properly propagated (issue 420)
- Allow falsey values in *Client.send_message()* and *Client.send_file()*.

2.3 v0.16.0

2.3.1 New Features

- Add *Channel.overwrites* to get all the permission overwrites of a channel.
- Add *Server.features* to get information about partnered servers.

2.3.2 Bug Fixes

- Timeout when waiting for offline members while triggering *on_ready()*.
  - The fact that we did not timeout caused a gigantic memory leak in the library that caused thousands of duplicate *Member* instances causing big memory spikes.
- Discard null sequences in the gateway.
  - The fact these were not discarded meant that *on_ready()* kept being called instead of *on_resumed()*. Since this has been corrected, in most cases *on_ready()* will be called once or twice with *on_resumed()* being called much more often.

2.4 v0.15.1

- Fix crash on duplicate or out of order reactions.

2.5 v0.15.0

2.5.1 New Features

- Rich Embeds for messages are now supported.
  - To do so, create your own *Embed* and pass the instance to the `embed` keyword argument to *Client.send_message()* or *Client.edit_message()*.
- Add *Client.clear_reactions()* to remove all reactions from a message.
- Add support for MESSAGE_REACTION_REMOVE_ALL event, under *on_reaction_clear()*.
- Add *Permissions.update()* and *PermissionOverwrite.update()* for bulk permission updates.
  - This allows you to use e.g. `p.update(read_messages=True, send_messages=False)` in a single line.
- Add *PermissionOverwrite.is_empty()* to check if the overwrite is empty (i.e. has no overwrites set explicitly as true or false).

For the command extension, the following changed:

- `Context` is no longer slotted to facilitate setting dynamic attributes.

2.6 v0.14.3

2.6.1 Bug Fixes

- Fix crash when dealing with MESSAGE_REACTION_REMOVE
- Fix incorrect buckets for reactions.

2.7 v0.14.2

2.7.1 New Features

- ***Client.wait_for_reaction()*** **now returns a namedtuple with `reaction` and `user` attributes.**
  - This is for better support in the case that `None` is returned since tuple unpacking can lead to issues.

2.7.2 Bug Fixes

- Fix bug that disallowed `None` to be passed for `emoji` parameter in *Client.wait_for_reaction()*.

2.8 v0.14.1

2.8.1 Bug fixes

- **Fix bug with *Reaction* not being visible at import.**
  - This was also breaking the documentation.

2.9 v0.14.0

This update adds new API features and a couple of bug fixes.

2.9.1 New Features

- Add support for Manage Webhooks permission under *Permissions.manage_webhooks*
- Add support for `around` argument in 3.5+ *Client.logs_from()*.
- **Add support for reactions.**
  - *Client.add_reaction()* to add a reactions
  - *Client.remove_reaction()* to remove a reaction.
  - *Client.get_reaction_users()* to get the users that reacted to a message.
  - *Permissions.add_reactions* permission bit support.
  - Two new events, *on_reaction_add()* and *on_reaction_remove()*.
  - *Message.reactions* to get reactions from a message.

  - *Client.wait_for_reaction()* to wait for a reaction from a user.

2.9.2 Bug Fixes

- Fix bug with Paginator still allowing lines that are too long.
- Fix the *Permissions.manage_emojis* bit being incorrect.

2.10 v0.13.0

This is a backwards compatible update with new features.

2.10.1 New Features

- Add the ability to manage emojis.
  - *Client.create_custom_emoji()* to create new emoji.
  - *Client.edit_custom_emoji()* to edit an old emoji.
  - *Client.delete_custom_emoji()* to delete a custom emoji.
- Add new *Permissions.manage_emojis* toggle.
  - This applies for *PermissionOverwrite* as well.
- Add new statuses for *Status*.
  - *Status.dnd* (aliased with *Status.do_not_disturb*) for Do Not Disturb.
  - *Status.invisible* for setting your status to invisible (please see the docs for a caveat).
- Deprecate *Client.change_status()*
  - Use *Client.change_presence()* instead for better more up to date functionality.
  - This method is subject for removal in a future API version.
- Add *Client.change_presence()* for changing your status with the new Discord API change.
  - This is the only method that allows changing your status to invisible or do not disturb.

2.10.2 Bug Fixes

- Paginator pages do not exceed their max_size anymore (issue 340)
- Do Not Disturb users no longer show up offline due to the new *Status* changes.

2.11 v0.12.0

This is a bug fix update that also comes with new features.

2.11.1 New Features

- Add custom emoji support.
  - Adds a new class to represent a custom Emoji named *Emoji*
  - Adds a utility generator function, *Client.get_all_emojis()*.
  - Adds a list of emojis on a server, *Server.emojis*.
  - Adds a new event, *on_server_emojis_update()*.
- Add new server regions to *ServerRegion*
  - *ServerRegion.eu_central* and *ServerRegion.eu_west*.
- Add support for new pinned system message under *MessageType.pins_add*.
- Add order comparisons for *Role* to allow it to be compared with regards to hierarchy.
  - This means that you can now do `role_a > role_b` etc to check if `role_b` is lower in the hierarchy.
- Add *Server.role_hierarchy* to get the server's role hierarchy.
- Add *Member.server_permissions* to get a member's server permissions without their channel specific overwrites.
- Add *Client.get_user_info()* to retrieve a user's info from their ID.
- Add a new `Player` property, `Player.error` to fetch the error that stopped the player.
  - To help with this change, a player's `after` function can now take a single parameter denoting the current player.
- Add support for server verification levels.
  - Adds a new enum called *VerificationLevel*.
  - This enum can be used in *Client.edit_server()* under the `verification_level` keyword argument.
  - Adds a new attribute in the server, *Server.verification_level*.
- Add *Server.voice_client* shortcut property for *Client.voice_client_in()*.
  - This is technically old (was added in v0.10.0) but was undocumented until v0.12.0.

For the command extension, the following are new:

- Add custom emoji converter.
- All default converters that can take IDs can now convert via ID.
- Add coroutine support for `Bot.command_prefix`.
- Add a method to reset command cooldown.

2.11.2 Bug Fixes

- Fix bug that caused the library to not work with the latest `websockets` library.
- Fix bug that leaked keep alive threads (issue 309)
- Fix bug that disallowed *ServerRegion* from being used in *Client.edit_server()*.
- Fix bug in *Channel.permissions_for()* that caused permission resolution to happen out of order.

- Fix bug in *Member.top_role* that did not account for same-position roles.

2.12 v0.11.0

This is a minor bug fix update that comes with a gateway update (v5 -> v6).

2.12.1 Breaking Changes

- `Permissions.change_nicknames` has been renamed to *Permissions.change_nickname* to match the UI.

2.12.2 New Features

- Add the ability to prune members via *Client.prune_members()*.
- Switch the websocket gateway version to v6 from v5. This allows the library to work with group DMs and 1-on-1 calls.
- Add *AppInfo.owner* attribute.
- Add *CallMessage* for group voice call messages.
- Add *GroupCall* for group voice call information.
- Add *Message.system_content* to get the system message.
- Add the remaining VIP servers and the Brazil servers into *ServerRegion* enum.
- Add `stderr` argument to *VoiceClient.create_ffmpeg_player()* to redirect stderr.
- The library now handles implicit permission resolution in *Channel.permissions_for()*.
- Add *Server.mfa_level* to query a server's 2FA requirement.
- Add *Permissions.external_emojis* permission.
- Add *Member.voice* attribute that refers to a *VoiceState*.
  - For backwards compatibility, the member object will have properties mirroring the old behaviour.

For the command extension, the following are new:

- Command cooldown system with the `cooldown` decorator.
- `UserInputError` exception for the hierarchy for user input related errors.

2.12.3 Bug Fixes

- *Client.email* is now saved when using a token for user accounts.
- Fix issue when removing roles out of order.
- Fix bug where discriminators would not update.
- Handle cases where `HEARTBEAT` opcode is received. This caused bots to disconnect seemingly randomly.

For the command extension, the following bug fixes apply:

- `Bot.check` decorator is actually a decorator not requiring parentheses.

- `Bot.remove_command` and `Group.remove_command` no longer throw if the command doesn't exist.
- Command names are no longer forced to be `lower()`.
- Fix a bug where Member and User converters failed to work in private message contexts.
- `HelpFormatter` now ignores hidden commands when deciding the maximum width.

2.13 v0.10.0

For breaking changes, see *Migrating to v0.10.0*. The breaking changes listed there will not be enumerated below. Since this version is rather a big departure from v0.9.2, this change log will be non-exhaustive.

2.13.1 New Features

- The library is now fully `asyncio` compatible, allowing you to write non-blocking code a lot more easily.
- The library now fully handles 429s and unconditionally retries on 502s.
- A new command extension module was added but is currently undocumented. Figuring it out is left as an exercise to the reader.
- Two new exception types, *Forbidden* and *NotFound* to denote permission errors or 404 errors.
- Added *Client.delete_invite()* to revoke invites.
- Added support for sending voice. Check *VoiceClient* for more details.
- Added *Client.wait_for_message()* coroutine to aid with follow up commands.
- Added *version_info* named tuple to check version info of the library.
- Login credentials are now cached to have a faster login experience. You can disable this by passing in `cache_auth=False` when constructing a *Client*.
- New utility function, *discord.utils.get()* to simplify retrieval of items based on attributes.
- All data classes now support `!=`, `==`, `hash(obj)` and `str(obj)`.
- Added *Client.get_bans()* to get banned members from a server.
- Added *Client.invites_from()* to get currently active invites in a server.
- Added *Server.me* attribute to get the *Member* version of *Client.user*.
- Most data classes now support a `hash(obj)` function to allow you to use them in `set` or `dict` classes or subclasses.
- Add *Message.clean_content()* to get a text version of the content with the user and channel mentioned changed into their names.
- Added a way to remove the messages of the user that just got banned in *Client.ban()*.
- Added *Client.wait_until_ready()* to facilitate easy creation of tasks that require the client cache to be ready.
- Added *Client.wait_until_login()* to facilitate easy creation of tasks that require the client to be logged in.
- Add *discord.Game* to represent any game with custom text to send to *Client.change_status()*.
- Add *Message.nonce* attribute.

- Add `Member.permissions_in()` as another way of doing *Channel.permissions_for()*.
- Add *Client.move_member()* to move a member to another voice channel.
- You can now create a server via *Client.create_server()*.
- Added *Client.edit_server()* to edit existing servers.
- Added *Client.server_voice_state()* to server mute or server deafen a member.
- If you are being rate limited, the library will now handle it for you.
- Add *on_member_ban()* and *on_member_unban()* events that trigger when a member is banned/unbanned.

2.13.2 Performance Improvements

- All data classes now use `__slots__` which greatly reduce the memory usage of things kept in cache.
- Due to the usage of `asyncio`, the CPU usage of the library has gone down significantly.
- A lot of the internal cache lists were changed into dictionaries to change the `O(n)` lookup into `O(1)`.
- Compressed READY is now on by default. This means if you're on a lot of servers (or maybe even a few) you would receive performance improvements by having to download and process less data.
- While minor, change regex from `\d+` to `[0-9]+` to avoid unnecessary unicode character lookups.

2.13.3 Bug Fixes

- Fix bug where guilds being updated did not edit the items in cache.
- Fix bug where `member.roles` were empty upon joining instead of having the `@everyone` role.
- Fix bug where *Role.is_everyone()* was not being set properly when the role was being edited.
- *Client.logs_from()* now handles cases where limit > 100 to sidestep the discord API limitation.
- Fix bug where a role being deleted would trigger a `ValueError`.
- Fix bug where *Permissions.kick_members()* and *Permissions.ban_members()* were flipped.
- Mentions are now triggered normally. This was changed due to the way discord handles it internally.
- Fix issue when a *Message* would attempt to upgrade a *Message.server* when the channel is a *Object*.
- Unavailable servers were not being added into cache, this has been corrected.

CHAPTER 3

Migrating to v0.10.0

v0.10.0 is one of the biggest breaking changes in the library due to massive fundamental changes in how the library operates.

The biggest major change is that the library has dropped support to all versions prior to Python 3.4.2. This was made to support `asyncio`, in which more detail can be seen in the corresponding issue. To reiterate this, the implication is that **python version 2.7 and 3.3 are no longer supported**.

Below are all the other major changes from v0.9.0 to v0.10.0.

3.1 Event Registration

All events before were registered using *`Client.event()`*. While this is still possible, the events must be decorated with `@asyncio.coroutine`.

Before:

```
@client.event
def on_message(message):
    pass
```

After:

```
@client.event
@asyncio.coroutine
def on_message(message):
    pass
```

Or in Python 3.5+:

```
@client.event
async def on_message(message):
    pass
```

Because there is a lot of typing, a utility decorator (*Client.async_event()*) is provided for easier registration. For example:

```
@client.async_event
def on_message(message):
    pass
```

Be aware however, that this is still a coroutine and your other functions that are coroutines must be decorated with `@asyncio.coroutine` or be `async def`.

3.2 Event Changes

Some events in v0.9.0 were considered pretty useless due to having no separate states. The main events that were changed were the `_update` events since previously they had no context on what was changed.

Before:

```
def on_channel_update(channel): pass
def on_member_update(member): pass
def on_status(member): pass
def on_server_role_update(role): pass
def on_voice_state_update(member): pass
def on_socket_raw_send(payload, is_binary): pass
```

After:

```
def on_channel_update(before, after): pass
def on_member_update(before, after): pass
def on_server_role_update(before, after): pass
def on_voice_state_update(before, after): pass
def on_socket_raw_send(payload): pass
```

Note that `on_status` was removed. If you want its functionality, use *on_member_update()*. See *Event Reference* for more information. Other removed events include `on_socket_closed`, `on_socket_receive`, and `on_socket_opened`.

3.3 Coroutines

The biggest change that the library went through is that almost every function in *Client* was changed to be a coroutine. Functions that are marked as a coroutine in the documentation must be awaited from or yielded from in order for the computation to be done. For example...

Before:

```
client.send_message(message.channel, 'Hello')
```

After:

```
yield from client.send_message(message.channel, 'Hello')

# or in python 3.5+
await client.send_message(message.channel, 'Hello')
```

In order for you to `yield from` or `await` a coroutine then your function must be decorated with `@asyncio.coroutine` or `async def`.

3.4 Iterables

For performance reasons, many of the internal data structures were changed into a dictionary to support faster lookup. As a consequence, this meant that some lists that were exposed via the API have changed into iterables and not sequences. In short, this means that certain attributes now only support iteration and not any of the sequence functions.

The affected attributes are as follows:

- *Client.servers*
- *Client.private_channels*
- *Server.channels*
- *Server.members*

Some examples of previously valid behaviour that is now invalid

```
if client.servers[0].name == "test":
    # do something
```

Since they are no longer `lists`, they no longer support indexing or any operation other than iterating. In order to get the old behaviour you should explicitly cast it to a list.

```
servers = list(client.servers)
# work with servers
```

Warning: Due to internal changes of the structure, the order you receive the data in is not in a guaranteed order.

3.5 Enumerations

Due to dropping support for versions lower than Python 3.4.2, the library can now use enumerations in places where it makes sense.

The common places where this was changed was in the server region, member status, and channel type.

Before:

```
server.region == 'us-west'
member.status == 'online'
channel.type == 'text'
```

After:

```
server.region == discord.ServerRegion.us_west
member.status = discord.Status.online
channel.type == discord.ChannelType.text
```

The main reason for this change was to reduce the use of finicky strings in the API as this could give users a false sense of power. More information can be found in the *Enumerations* page.

3.6 Properties

A lot of function calls that returned constant values were changed into Python properties for ease of use in format strings.

The following functions were changed into properties:

Before	After
`User.avatar_url()`	*`User.avatar_url`*
`User.mention()`	*`User.mention`*
`Channel.mention()`	*`Channel.mention`*
`Channel.is_default_channel()`	*`Channel.is_default`*
`Role.is_everyone()`	*`Role.is_everyone`*
`Server.get_default_role()`	*`Server.default_role`*
`Server.icon_url()`	*`Server.icon_url`*
`Server.get_default_channel()`	*`Server.default_channel`*
`Message.get_raw_mentions()`	*`Message.raw_mentions`*
`Message.get_raw_channel_mentions()`	*`Message.raw_channel_mentions`*

3.7 Member Management

Functions that involved banning and kicking were changed.

Before	After
`Client.ban(server, user)`	`Client.ban(member)`
`Client.kick(server, user)`	`Client.kick(member)`

3.8 Renamed Functions

Functions have been renamed.

Before	After
`Client.set_channel_permissions`	*`Client.edit_channel_permissions()`*

All the *`Permissions`* related attributes have been renamed and the *can_* prefix has been dropped. So for example, `can_manage_messages` has become `manage_messages`.

3.9 Forced Keyword Arguments

Since 3.0+ of Python, we can now force questions to take in forced keyword arguments. A keyword argument is when you explicitly specify the name of the variable and assign to it, for example: `foo(name='test')`. Due to this support, some functions in the library were changed to force things to take said keyword arguments. This is to reduce errors of knowing the argument order and the issues that could arise from them.

The following parameters are now exclusively keyword arguments:

- ***`Client.send_message()`***

        - tts
    - *Client.logs_from()*
        - before
        - after
    - *Client.edit_channel_permissions()*
        - allow
        - deny

In the documentation you can tell if a function parameter is a forced keyword argument if it is after `\*,` in the function signature.

3.10 Running the Client

In earlier versions of discord.py, `client.run()` was a blocking call to the main thread that called it. In v0.10.0 it is still a blocking call but it handles the event loop for you. However, in order to do that you must pass in your credentials to *Client.run()*.

Basically, before:

```
client.login('token')
client.run()
```

After:

```
client.run('token')
```

Warning: Like in the older `Client.run` function, the newer one must be the one of the last functions to call. This is because the function is **blocking**. Registering events or doing anything after *Client.run()* will not execute until the function returns.

This is a utility function that abstracts the event loop for you. There's no need for the run call to be blocking and out of your control. Indeed, if you want control of the event loop then doing so is quite straightforward:

```
import discord
import asyncio

client = discord.Client()

@asyncio.coroutine
def main_task():
    yield from client.login('token')
    yield from client.connect()

loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(main_task())
except:
    loop.run_until_complete(client.logout())
finally:
    loop.close()
```

CHAPTER 4

API Reference

The following section outlines the API of discord.py.

Note: This module uses the Python logging module to log diagnostic and errors in an output independent way. If the logging module is not configured, these logs will not be output anywhere. See *Setting Up Logging* for more information on how to set up and use the logging module with discord.py.

4.1 Version Related Info

There are two main ways to query version information about the library.

`discord.`**`version_info`**
A named tuple that is similar to sys.version_info.

Just like sys.version_info the valid values for `releaselevel` are 'alpha', 'beta', 'candidate' and 'final'.

`discord.`**`__version__`**
A string representation of the version. e.g. `'0.10.0-alpha0'`.

4.2 Client

class `discord.`**`Client`**(*, *loop=None*, ***options*)
Represents a client connection that connects to Discord. This class is used to interact with the Discord WebSocket and API.

A number of options can be passed to the *`Client`*.

Parameters

- **max_messages** (*Optional[int]*) – The maximum number of messages to store in *messages*. This defaults to 5000. Passing in *None* or a value less than 100 will use the default instead of the passed in value.
- **loop** (*Optional[event loop]*) – The event loop to use for asynchronous operations. Defaults to `None`, in which case the default event loop is used via `asyncio.get_event_loop()`.
- **cache_auth** (*Optional[bool]*) – Indicates if *login()* should cache the authentication tokens. Defaults to `True`. The method in which the cache is written is done by writing to disk to a temporary directory.
- **connector** (*aiohttp.BaseConnector*) – The connector to use for connection pooling. Useful for proxies, e.g. with a ProxyConnector.
- **shard_id** (*Optional[int]*) – Integer starting at 0 and less than shard_count.
- **shard_count** (*Optional[int]*) – The total number of shards.

user
Optional[*User*] – Represents the connected client. None if not logged in.

voice_clients
iterable of *VoiceClient* – Represents a list of voice connections. To connect to voice use *join_voice_channel()*. To query the voice connection state use *is_voice_connected()*.

servers
iterable of *Server* – The servers that the connected client is a member of.

private_channels
iterable of *PrivateChannel* – The private channels that the connected client is participating on.

messages
A deque of *Message* that the client has received from all servers and private messages. The number of messages stored in this deque is controlled by the `max_messages` parameter.

email
The email used to login. This is only set if login is successful, otherwise it's None.

ws
The websocket gateway the client is currently connected to. Could be None.

loop
The event loop that the client uses for HTTP requests and websocket operations.

on_error(*event_method*, **args*, ***kwargs*)
This function is a *coroutine*.

The default error handler provided by the client.

By default this prints to `sys.stderr` however it could be overridden to have a different implementation. Check *discord.on_error()* for more details.

login(**args*, ***kwargs*)
This function is a *coroutine*.

Logs in the client with the specified credentials.

This function can be used in two different ways.

```
await client.login('token')

# or
```

```
await client.login('email', 'password')
```

More than 2 parameters or less than 1 parameter raises a `TypeError`.

Parameters **bot** (*bool*) – Keyword argument that specifies if the account logging on is a bot token or not. Only useful for logging in with a static token. Ignored for the email and password combo. Defaults to `True`.

Raises

- *LoginFailure* – The wrong credentials are passed.
- *HTTPException* – An unknown HTTP related error occurred, usually when it isn't 200 or the known incorrect credentials passing status code.
- `TypeError` – The incorrect number of parameters is passed.

logout()

This function is a *coroutine*.

Logs out of Discord and closes all connections.

connect()

This function is a *coroutine*.

Creates a websocket connection and lets the websocket listen to messages from discord.

Raises

- *GatewayNotFound* – If the gateway to connect to discord is not found. Usually if this is thrown then there is a discord API outage.
- *ConnectionClosed* – The websocket connection has been terminated.

close()

This function is a *coroutine*.

Closes the connection to discord.

start(**args*, ***kwargs*)

This function is a *coroutine*.

A shorthand coroutine for *login()* + *connect()*.

run(**args*, ***kwargs*)

A blocking call that abstracts away the event loop initialisation from you.

If you want more control over the event loop then this function should not be used. Use *start()* coroutine or *connect()* + *login()*.

Roughly Equivalent to:

```
try:
    loop.run_until_complete(start(*args, **kwargs))
except KeyboardInterrupt:
    loop.run_until_complete(logout())
    # cancel all tasks lingering
finally:
    loop.close()
```

> **Warning:** This function must be the last function to call due to the fact that it is blocking. That means that registration of events or anything being called after this function call will not execute until it returns.

is_logged_in
bool – Indicates if the client has logged in successfully.

is_closed
bool – Indicates if the websocket connection is closed.

get_channel(*id*)
Returns a *Channel* or *PrivateChannel* with the following ID. If not found, returns None.

get_server(*id*)
Returns a *Server* with the given ID. If not found, returns None.

get_all_emojis()
Returns a generator with every *Emoji* the client can see.

get_all_channels()
A generator that retrieves every *Channel* the client can 'access'.

This is equivalent to:

```
for server in client.servers:
    for channel in server.channels:
        yield channel
```

> **Note:** Just because you receive a *Channel* does not mean that you can communicate in said channel. *Channel.permissions_for()* should be used for that.

get_all_members()
Returns a generator with every *Member* the client can see.

This is equivalent to:

```
for server in client.servers:
    for member in server.members:
        yield member
```

wait_until_ready()
This function is a *coroutine*.

This coroutine waits until the client is all ready. This could be considered another way of asking for *discord.on_ready()* except meant for your own background tasks.

wait_until_login()
This function is a *coroutine*.

This coroutine waits until the client is logged on successfully. This is different from waiting until the client's state is all ready. For that check *discord.on_ready()* and *wait_until_ready()*.

wait_for_message(*timeout=None*, *, *author=None*, *channel=None*, *content=None*, *check=None*)
This function is a *coroutine*.

Waits for a message reply from Discord. This could be seen as another *discord.on_message()* event outside of the actual event. This could also be used for follow-ups and easier user interactions.

The keyword arguments passed into this function are combined using the logical and operator. The `check` keyword argument can be used to pass in more complicated checks and must be a regular function (not a coroutine).

The `timeout` parameter is passed into asyncio.wait_for. By default, it does not timeout. Instead of throwing `asyncio.TimeoutError` the coroutine catches the exception and returns `None` instead of a *Message*.

If the `check` predicate throws an exception, then the exception is propagated.

This function returns the **first message that meets the requirements**.

Examples

Basic example:

```
@client.event
async def on_message(message):
    if message.content.startswith('$greet'):
        await client.send_message(message.channel, 'Say hello')
        msg = await client.wait_for_message(author=message.author, content=
↪'hello')
        await client.send_message(message.channel, 'Hello.')
```

Asking for a follow-up question:

```
@client.event
async def on_message(message):
    if message.content.startswith('$start'):
        await client.send_message(message.channel, 'Type $stop 4 times.')
        for i in range(4):
            msg = await client.wait_for_message(author=message.author,␣
↪content='$stop')
            fmt = '{} left to go...'
            await client.send_message(message.channel, fmt.format(3 - i))

        await client.send_message(message.channel, 'Good job!')
```

Advanced filters using `check`:

```
@client.event
async def on_message(message):
    if message.content.startswith('$cool'):
        await client.send_message(message.channel, 'Who is cool? Type $name␣
↪namehere')

        def check(msg):
            return msg.content.startswith('$name')

        message = await client.wait_for_message(author=message.author,␣
↪check=check)
        name = message.content[len('$name'):].strip()
        await client.send_message(message.channel, '{} is cool indeed'.
↪format(name))
```

Parameters

- **timeout** (*float*) – The number of seconds to wait before returning `None`.

- **author** (*Member* or *User*) – The author the message must be from.
- **channel** (*Channel* or *PrivateChannel* or *Object*) – The channel the message must be from.
- **content** (*str*) – The exact content the message must have.
- **check** (*function*) – A predicate for other complicated checks. The predicate must take a *Message* as its only parameter.

Returns The message that you requested for.

Return type *Message*

wait_for_reaction(*emoji=None*, ***, *user=None*, *timeout=None*, *message=None*, *check=None*)

This function is a *coroutine*.

Waits for a message reaction from Discord. This is similar to *wait_for_message()* and could be seen as another *on_reaction_add()* event outside of the actual event. This could be used for follow up situations.

Similar to *wait_for_message()*, the keyword arguments are combined using logical AND operator. The `check` keyword argument can be used to pass in more complicated checks and must a regular function taking in two arguments, `(reaction, user)`. It must not be a coroutine.

The `timeout` parameter is passed into asyncio.wait_for. By default, it does not timeout. Instead of throwing `asyncio.TimeoutError` the coroutine catches the exception and returns `None` instead of a the `(reaction, user)` tuple.

If the `check` predicate throws an exception, then the exception is propagated.

The `emoji` parameter can be either a *Emoji*, a `str` representing an emoji, or a sequence of either type. If the `emoji` parameter is a sequence then the first reaction emoji that is in the list is returned. If `None` is passed then the first reaction emoji used is returned.

This function returns the **first reaction that meets the requirements**.

Examples

Basic Example:

```
@client.event
async def on_message(message):
    if message.content.startswith('$react'):
        msg = await client.send_message(message.channel, 'React with thumbs␣
↪up or thumbs down.')
        res = await client.wait_for_reaction(['', ''], message=msg)
        await client.send_message(message.channel, '{0.user} reacted with {0.
↪reaction.emoji}!'.format(res))
```

Checking for reaction emoji regardless of skin tone:

```
@client.event
async def on_message(message):
    if message.content.startswith('$react'):
        msg = await client.send_message(message.channel, 'React with thumbs␣
↪up or thumbs down.')

        def check(reaction, user):
            e = str(reaction.emoji)
```

```
            return e.startswith(('', ''))

        res = await client.wait_for_reaction(message=msg, check=check)
        await client.send_message(message.channel, '{0.user} reacted with {0.
↪reaction.emoji}!'.format(res))
```

Parameters

- **timeout** (*float*) – The number of seconds to wait before returning `None`.
- **user** (*Member* or *User*) – The user the reaction must be from.
- **emoji** (str or *Emoji* or sequence) – The emoji that we are waiting to react with.
- **message** (*Message*) – The message that we want the reaction to be from.
- **check** (*function*) – A predicate for other complicated checks. The predicate must take `(reaction, user)` as its two parameters, which `reaction` being a *Reaction* and `user` being either a *User* or a *Member*.

Returns A namedtuple with attributes `reaction` and `user` similar to *on_reaction_add()*.

Return type namedtuple

event (*coro*)

A decorator that registers an event to listen to.

You can find more info about the events on the *documentation below*.

The events must be a *coroutine*, if not, *ClientException* is raised.

Examples

Using the basic *event()* decorator:

```
@client.event
@asyncio.coroutine
def on_ready():
    print('Ready!')
```

Saving characters by using the *async_event()* decorator:

```
@client.async_event
def on_ready():
    print('Ready!')
```

async_event (*coro*)

A shorthand decorator for `asyncio.coroutine` + *event()*.

start_private_message (*user*)

This function is a *coroutine*.

Starts a private message with the user. This allows you to *send_message()* to the user.

Note: This method should rarely be called as *send_message()* does it automatically for you.

Parameters **user** (*User*) – The user to start the private message with.

Raises

- *HTTPException* – The request failed.
- *InvalidArgument* – The user argument was not of *User*.

add_reaction(*message*, *emoji*)
This function is a *coroutine*.

Add a reaction to the given message.

The message must be a *Message* that exists. emoji may be a unicode emoji, or a custom server *Emoji*.

Parameters

- **message** (*Message*) – The message to react to.
- **emoji** (*Emoji* or str) – The emoji to react with.

Raises

- *HTTPException* – Adding the reaction failed.
- *Forbidden* – You do not have the proper permissions to react to the message.
- *NotFound* – The message or emoji you specified was not found.
- *InvalidArgument* – The message or emoji parameter is invalid.

remove_reaction(*message*, *emoji*, *member*)
This function is a *coroutine*.

Remove a reaction by the member from the given message.

If member != server.me, you need Manage Messages to remove the reaction.

The message must be a *Message* that exists. emoji may be a unicode emoji, or a custom server *Emoji*.

Parameters

- **message** (*Message*) – The message.
- **emoji** (*Emoji* or str) – The emoji to remove.
- **member** (*Member*) – The member for which to delete the reaction.

Raises

- *HTTPException* – Removing the reaction failed.
- *Forbidden* – You do not have the proper permissions to remove the reaction.
- *NotFound* – The message or emoji you specified was not found.
- *InvalidArgument* – The message or emoji parameter is invalid.

get_reaction_users(*reaction*, *limit=100*, *after=None*)
This function is a *coroutine*.

Get the users that added a reaction to a message.

Parameters

- **reaction** (*Reaction*) – The reaction to retrieve users for.
- **limit** (*int*) – The maximum number of results to return.
- **after** (*Member* or *Object*) – For pagination, reactions are sorted by member.

Raises

- *HTTPException* – Getting the users for the reaction failed.
- *NotFound* – The message or emoji you specified was not found.
- *InvalidArgument* – The reaction parameter is invalid.

clear_reactions(*message*)
This function is a *coroutine*.

Removes all the reactions from a given message.

You need Manage Messages permission to use this.

Parameters **message** (*Message*) – The message to remove all reactions from.

Raises

- *HTTPException* – Removing the reactions failed.
- *Forbidden* – You do not have the proper permissions to remove all the reactions.

send_message(*destination*, *content=None*, ***, *tts=False*, *embed=None*)
This function is a *coroutine*.

Sends a message to the destination given with the content given.

The destination could be a *Channel*, *PrivateChannel* or *Server*. For convenience it could also be a *User*. If it's a *User* or *PrivateChannel* then it sends the message via private message, otherwise it sends the message to the channel. If the destination is a *Server* then it's equivalent to calling *Server.default_channel* and sending it there.

If it is a *Object* instance then it is assumed to be the destination ID. The destination ID is a *channel* so passing in a user ID will not be a valid destination.

Changed in version 0.9.0: `str` being allowed was removed and replaced with *Object*.

The content must be a type that can convert to a string through `str(content)`. If the content is set to `None` (the default), then the `embed` parameter must be provided.

If the `embed` parameter is provided, it must be of type *Embed* and it must be a rich embed type.

Parameters

- **destination** – The location to send the message.
- **content** – The content of the message to send. If this is missing, then the `embed` parameter must be present.
- **tts** (*bool*) – Indicates if the message should be sent using text-to-speech.
- **embed** (*Embed*) – The rich embed for the content.

Raises

- *HTTPException* – Sending the message failed.
- *Forbidden* – You do not have the proper permissions to send the message.
- *NotFound* – The destination was not found and hence is invalid.
- *InvalidArgument* – The destination parameter is invalid.

Examples

Sending a regular message:

```
await client.send_message(message.channel, 'Hello')
```

Sending a TTS message:

```
await client.send_message(message.channel, 'Goodbye.', tts=True)
```

Sending an embed message:

```
em = discord.Embed(title='My Embed Title', description='My Embed Content.',␣
↪colour=0xDEADBF)
em.set_author(name='Someone', icon_url=client.user.default_avatar_url)
await client.send_message(message.channel, embed=em)
```

Returns The message that was sent.

Return type *Message*

send_typing(*destination*)

This function is a *coroutine*.

Send a *typing* status to the destination.

Typing status will go away after 10 seconds, or after a message is sent.

The destination parameter follows the same rules as *send_message()*.

Parameters **destination** – The location to send the typing update.

send_file(*destination*, *fp*, ***, *filename=None*, *content=None*, *tts=False*)

This function is a *coroutine*.

Sends a message to the destination given with the file given.

The destination parameter follows the same rules as *send_message()*.

The `fp` parameter should be either a string denoting the location for a file or a *file-like object*. The *file-like object* passed is **not closed** at the end of execution. You are responsible for closing it yourself.

Note: If the file-like object passed is opened via `open` then the modes 'rb' should be used.

The `filename` parameter is the filename of the file. If this is not given then it defaults to `fp.name` or if `fp` is a string then the `filename` will default to the string given. You can overwrite this value by passing this in.

Parameters

- **destination** – The location to send the message.
- **fp** – The *file-like object* or file path to send.
- **filename** (*str*) – The filename of the file. Defaults to `fp.name` if it's available.
- **content** – The content of the message to send along with the file. This is forced into a string by a `str(content)` call.
- **tts** (*bool*) – If the content of the message should be sent with TTS enabled.

Raises *HTTPException* – Sending the file failed.

Returns The message sent.

Return type *Message*

delete_message(*message*)
This function is a *coroutine*.

Deletes a *Message*.

Your own messages could be deleted without any proper permissions. However to delete other people's messages, you need the proper permissions to do so.

Parameters **message** (*Message*) – The message to delete.

Raises

- *Forbidden* – You do not have proper permissions to delete the message.
- *HTTPException* – Deleting the message failed.

delete_messages(*messages*)
This function is a *coroutine*.

Deletes a list of messages. This is similar to *delete_message()* except it bulk deletes multiple messages.

The channel to check where the message is deleted from is handled via the first element of the iterable's `.channel.id` attributes. If the channel is not consistent throughout the entire sequence, then an *HTTPException* will be raised.

Usable only by bot accounts.

Parameters **messages** (iterable of *Message*) – An iterable of messages denoting which ones to bulk delete.

Raises

- *ClientException* – The number of messages to delete is less than 2 or more than 100.
- *Forbidden* – You do not have proper permissions to delete the messages or you're not using a bot account.
- *HTTPException* – Deleting the messages failed.

purge_from(*channel*, *, *limit=100*, *check=None*, *before=None*, *after=None*, *around=None*)
This function is a *coroutine*.

Purges a list of messages that meet the criteria given by the predicate `check`. If a `check` is not provided then all messages are deleted without discrimination.

You must have Manage Messages permission to delete messages even if they are your own. The Read Message History permission is also needed to retrieve message history.

Usable only by bot accounts.

Parameters

- **channel** (*Channel*) – The channel to purge from.
- **limit** (*int*) – The number of messages to search through. This is not the number of messages that will be deleted, though it can be.
- **check** (*predicate*) – The function used to check if a message should be deleted. It must take a *Message* as its sole parameter.

- **before** (*Message* or *datetime*) – The message or date before which all deleted messages must be. If a date is provided it must be a timezone-naive datetime representing UTC time.
- **after** (*Message* or *datetime*) – The message or date after which all deleted messages must be. If a date is provided it must be a timezone-naive datetime representing UTC time.
- **around** (*Message* or *datetime*) – The message or date around which all deleted messages must be. If a date is provided it must be a timezone-naive datetime representing UTC time.

Raises

- *Forbidden* – You do not have proper permissions to do the actions required or you're not using a bot account.
- *HTTPException* – Purging the messages failed.

Examples

Deleting bot's messages

```
def is_me(m):
    return m.author == client.user

deleted = await client.purge_from(channel, limit=100, check=is_me)
await client.send_message(channel, 'Deleted {} message(s)'.
↪format(len(deleted)))
```

Returns The list of messages that were deleted.

Return type list

edit_message(*message*, *new_content=None*, *, *embed=None*)

This function is a *coroutine*.

Edits a *Message* with the new message content.

The new_content must be able to be transformed into a string via `str(new_content)`.

If the `new_content` is not provided, then `embed` must be provided, which must be of type *Embed*.

The *Message* object is not directly modified afterwards until the corresponding WebSocket event is received.

Parameters

- **message** (*Message*) – The message to edit.
- **new_content** – The new content to replace the message with.
- **embed** (*Embed*) – The new embed to replace the original embed with.

Raises *HTTPException* – Editing the message failed.

Returns The new edited message.

Return type *Message*

get_message(*channel*, *id*)

This function is a *coroutine*.

Retrieves a single *Message* from a *Channel*.

This can only be used by bot accounts.

Parameters

- **channel** (*Channel* or *PrivateChannel*) – The text channel to retrieve the message from.
- **id** (*str*) – The message ID to look for.

Returns The message asked for.

Return type *Message*

Raises

- *NotFound* – The specified channel or message was not found.
- *Forbidden* – You do not have the permissions required to get a message.
- *HTTPException* – Retrieving the message failed.

pin_message(*message*)
This function is a *coroutine*.

Pins a message. You must have Manage Messages permissions to do this in a non-private channel context.

Parameters **message** (*Message*) – The message to pin.

Raises

- *Forbidden* – You do not have permissions to pin the message.
- *NotFound* – The message or channel was not found.
- *HTTPException* – Pinning the message failed, probably due to the channel having more than 50 pinned messages.

unpin_message(*message*)
This function is a *coroutine*.

Unpins a message. You must have Manage Messages permissions to do this in a non-private channel context.

Parameters **message** (*Message*) – The message to unpin.

Raises

- *Forbidden* – You do not have permissions to unpin the message.
- *NotFound* – The message or channel was not found.
- *HTTPException* – Unpinning the message failed.

pins_from(*channel*)
This function is a *coroutine*.

Returns a list of *Message* that are currently pinned for the specified *Channel* or *PrivateChannel*.

Parameters **channel** (*Channel* or *PrivateChannel*) – The channel to look through pins for.

Raises

- *NotFound* – The channel was not found.
- *HTTPException* – Retrieving the pinned messages failed.

logs_from(*channel*, *limit=100*, *, *before=None*, *after=None*, *around=None*, *reverse=False*)
This function is a *coroutine*.

This coroutine returns a generator that obtains logs from a specified channel.

Parameters

- **channel** (*Channel* or *PrivateChannel*) – The channel to obtain the logs from.
- **limit** (*int*) – The number of messages to retrieve.
- **before** (*Message* or *datetime*) – The message or date before which all returned messages must be. If a date is provided it must be a timezone-naive datetime representing UTC time.
- **after** (*Message* or *datetime*) – The message or date after which all returned messages must be. If a date is provided it must be a timezone-naive datetime representing UTC time.
- **around** (*Message* or *datetime*) – The message or date around which all returned messages must be. If a date is provided it must be a timezone-naive datetime representing UTC time.

Raises

- *Forbidden* – You do not have permissions to get channel logs.
- *NotFound* – The channel you are requesting for doesn't exist.
- *HTTPException* – The request to get logs failed.

Yields *Message* – The message with the message data parsed.

Examples

Basic logging:

```
logs = yield from client.logs_from(channel)
for message in logs:
    if message.content.startswith('!hello'):
        if message.author == client.user:
            yield from client.edit_message(message, 'goodbye')
```

Python 3.5 Usage

```
counter = 0
async for message in client.logs_from(channel, limit=500):
    if message.author == client.user:
        counter += 1
```

request_offline_members(*server*)
This function is a *coroutine*.

Requests previously offline members from the server to be filled up into the *Server.members* cache. This function is usually not called.

When the client logs on and connects to the websocket, Discord does not provide the library with offline members if the number of members in the server is larger than 250. You can check if a server is large if *Server.large* is `True`.

Parameters **server** (*Server* or iterable) – The server to request offline members for. If this parameter is a iterable then it is interpreted as an iterator of servers to request offline members for.

kick(*member*)
This function is a *coroutine*.

Kicks a *Member* from the server they belong to.

> **Warning:** This function kicks the *Member* based on the server it belongs to, which is accessed via *Member.server*. So you must have the proper permissions in that server.

Parameters **member** (*Member*) – The member to kick from their server.

Raises

- *Forbidden* – You do not have the proper permissions to kick.
- *HTTPException* – Kicking failed.

ban(*member*, *delete_message_days=1*)
This function is a *coroutine*.

Bans a *Member* from the server they belong to.

> **Warning:** This function bans the *Member* based on the server it belongs to, which is accessed via *Member.server*. So you must have the proper permissions in that server.

Parameters

- **member** (*Member*) – The member to ban from their server.
- **delete_message_days** (*int*) – The number of days worth of messages to delete from the user in the server. The minimum is 0 and the maximum is 7.

Raises

- *Forbidden* – You do not have the proper permissions to ban.
- *HTTPException* – Banning failed.

unban(*server*, *user*)
This function is a *coroutine*.

Unbans a *User* from the server they are banned from.

Parameters

- **server** (*Server*) – The server to unban the user from.
- **user** (*User*) – The user to unban.

Raises

- *Forbidden* – You do not have the proper permissions to unban.
- *HTTPException* – Unbanning failed.

server_voice_state(*member*, *, *mute=None*, *deafen=None*)
This function is a *coroutine*.

Server mutes or deafens a specific *Member*.

> **Warning:** This function mutes or un-deafens the *Member* based on the server it belongs to, which is accessed via *Member.server*. So you must have the proper permissions in that server.

Parameters

- **member** (*Member*) – The member to unban from their server.
- **mute** (*Optional[bool]*) – Indicates if the member should be server muted or un-muted.
- **deafen** (*Optional[bool]*) – Indicates if the member should be server deafened or un-deafened.

Raises

- *Forbidden* – You do not have the proper permissions to deafen or mute.
- *HTTPException* – The operation failed.

edit_profile (*password=None*, ***fields*)

This function is a *coroutine*.

Edits the current profile of the client.

If a bot account is used then the password field is optional, otherwise it is required.

The *Client.user* object is not modified directly afterwards until the corresponding WebSocket event is received.

Note: To upload an avatar, a *bytes-like object* must be passed in that represents the image being uploaded. If this is done through a file then the file must be opened via `open('some_filename', 'rb')` and the *bytes-like object* is given through the use of `fp.read()`.

The only image formats supported for uploading is JPEG and PNG.

Parameters

- **password** (*str*) – The current password for the client's account. Not used for bot accounts.
- **new_password** (*str*) – The new password you wish to change to.
- **email** (*str*) – The new email you wish to change to.
- **username** (*str*) – The new username you wish to change to.
- **avatar** (*bytes*) – A *bytes-like object* representing the image to upload. Could be `None` to denote no avatar.

Raises

- *HTTPException* – Editing your profile failed.
- *InvalidArgument* – Wrong image format passed for `avatar`.
- *ClientException* – Password is required for non-bot accounts.

change_status (*game=None*, *idle=False*)

This function is a *coroutine*.

Changes the client's status.

The game parameter is a Game object (not a string) that represents a game being played currently.

The idle parameter is a boolean parameter that indicates whether the client should go idle or not.

Deprecated since version v0.13.0: Use *change_presence()* instead.

Parameters

- **game** (Optional[*Game*]) – The game being played. None if no game is being played.
- **idle** (*bool*) – Indicates if the client should go idle.

Raises *InvalidArgument* – If the `game` parameter is not *Game* or None.

change_presence(*, *game=None*, *status=None*, *afk=False*)
This function is a *coroutine*.

Changes the client's presence.

The game parameter is a Game object (not a string) that represents a game being played currently.

Parameters

- **game** (Optional[*Game*]) – The game being played. None if no game is being played.
- **status** (Optional[*Status*]) – Indicates what status to change to. If None, then *Status.online* is used.
- **afk** (*bool*) – Indicates if you are going AFK. This allows the discord client to know how to handle push notifications better for you in case you are actually idle and not lying.

Raises *InvalidArgument* – If the `game` parameter is not *Game* or None.

change_nickname(*member*, *nickname*)
This function is a *coroutine*.

Changes a member's nickname.

You must have the proper permissions to change someone's (or your own) nickname.

Parameters

- **member** (*Member*) – The member to change the nickname for.
- **nickname** (*Optional[str]*) – The nickname to change it to. `None` to remove the nickname.

Raises

- *Forbidden* – You do not have permissions to change the nickname.
- *HTTPException* – Changing the nickname failed.

edit_channel(*channel*, ***options*)
This function is a *coroutine*.

Edits a *Channel*.

You must have the proper permissions to edit the channel.

To move the channel's position use *move_channel()* instead.

The *Channel* object is not directly modified afterwards until the corresponding WebSocket event is received.

Parameters

- **channel** (*Channel*) – The channel to update.

- **name** (*str*) – The new channel name.
- **topic** (*str*) – The new channel's topic.
- **bitrate** (*int*) – The new channel's bitrate. Voice only.
- **user_limit** (*int*) – The new channel's user limit. Voice only.

Raises

- *Forbidden* – You do not have permissions to edit the channel.
- *HTTPException* – Editing the channel failed.

move_channel (*channel*, *position*)

This function is a *coroutine*.

Moves the specified *Channel* to the given position in the GUI. Note that voice channels and text channels have different position values.

The *Channel* object is not directly modified afterwards until the corresponding WebSocket event is received.

> **Warning:** *Object* instances do not work with this function.

Parameters

- **channel** (*Channel*) – The channel to change positions of.
- **position** (*int*) – The position to insert the channel to.

Raises

- *InvalidArgument* – If position is less than 0 or greater than the number of channels.
- *Forbidden* – You do not have permissions to change channel order.
- *HTTPException* – If moving the channel failed, or you are of too low rank to move the channel.

create_channel (*server*, *name*, **overwrites*, *type=None*)

This function is a *coroutine*.

Creates a *Channel* in the specified *Server*.

Note that you need the proper permissions to create the channel.

The `overwrites` argument list can be used to create a 'secret' channel upon creation. A namedtuple of `ChannelPermissions` is exposed to create a channel-specific permission overwrite in a more self-documenting matter. You can also use a regular tuple of `(target, overwrite)` where the `overwrite` expected has to be of type *PermissionOverwrite*.

Examples

Creating a voice channel:

```
await client.create_channel(server, 'Voice', type=discord.ChannelType.voice)
```

Creating a 'secret' text channel:

```
everyone_perms = discord.PermissionOverwrite(read_messages=False)
my_perms = discord.PermissionOverwrite(read_messages=True)

everyone = discord.ChannelPermissions(target=server.default_role,␣
↪overwrite=everyone_perms)
mine = discord.ChannelPermissions(target=server.me, overwrite=my_perms)
await client.create_channel(server, 'secret', everyone, mine)
```

Or in a more 'compact' way:

```
everyone = discord.PermissionOverwrite(read_messages=False)
mine = discord.PermissionOverwrite(read_messages=True)
await client.create_channel(server, 'secret', (server.default_role, everyone),
↪ (server.me, mine))
```

Parameters

- **server** (*Server*) – The server to create the channel in.
- **name** (*str*) – The channel's name.
- **type** (*ChannelType*) – The type of channel to create. Defaults to *ChannelType.text*.
- **overwrites** – An argument list of channel specific overwrites to apply on the channel on creation. Useful for creating 'secret' channels.

Raises

- *Forbidden* – You do not have the proper permissions to create the channel.
- *NotFound* – The server specified was not found.
- *HTTPException* – Creating the channel failed.
- *InvalidArgument* – The permission overwrite array is not in proper form.

Returns The channel that was just created. This channel is different than the one that will be added in cache.

Return type *Channel*

delete_channel(*channel*)

This function is a *coroutine*.

Deletes a *Channel*.

In order to delete the channel, the client must have the proper permissions in the server the channel belongs to.

Parameters **channel** (*Channel*) – The channel to delete.

Raises

- *Forbidden* – You do not have proper permissions to delete the channel.
- *NotFound* – The specified channel was not found.
- *HTTPException* – Deleting the channel failed.

leave_server(*server*)

This function is a *coroutine*.

Leaves a *Server*.

Note: You cannot leave the server that you own, you must delete it instead via *delete_server()*.

Parameters **server** (*Server*) – The server to leave.

Raises *HTTPException* – If leaving the server failed.

delete_server(*server*)
This function is a *coroutine*.

Deletes a *Server*. You must be the server owner to delete the server.

Parameters **server** (*Server*) – The server to delete.

Raises

- *HTTPException* – If deleting the server failed.
- *Forbidden* – You do not have permissions to delete the server.

create_server(*name*, *region=None*, *icon=None*)
This function is a *coroutine*.

Creates a *Server*.

Bot accounts generally are not allowed to create servers. See Discord's official documentation for more info.

Parameters

- **name** (*str*) – The name of the server.
- **region** (*ServerRegion*) – The region for the voice communication server. Defaults to *ServerRegion.us_west*.
- **icon** (*bytes*) – The *bytes-like* object representing the icon. See *edit_profile()* for more details on what is expected.

Raises

- *HTTPException* – Server creation failed.
- *InvalidArgument* – Invalid icon image format given. Must be PNG or JPG.

Returns The server created. This is not the same server that is added to cache.

Return type *Server*

edit_server(*server*, ***fields*)
This function is a *coroutine*.

Edits a *Server*.

You must have the proper permissions to edit the server.

The *Server* object is not directly modified afterwards until the corresponding WebSocket event is received.

Parameters

- **server** (*Server*) – The server to edit.
- **name** (*str*) – The new name of the server.
- **icon** (*bytes*) – A *bytes-like* object representing the icon. See *edit_profile()* for more details. Could be `None` to denote no icon.

- **splash** (*bytes*) – A *bytes-like* object representing the invite splash. See *edit_profile()* for more details. Could be `None` to denote no invite splash. Only available for partnered servers with `INVITE_SPLASH` feature.
- **region** (*ServerRegion*) – The new region for the server's voice communication.
- **afk_channel** (Optional[*Channel*]) – The new channel that is the AFK channel. Could be `None` for no AFK channel.
- **afk_timeout** (*int*) – The number of seconds until someone is moved to the AFK channel.
- **owner** (*Member*) – The new owner of the server to transfer ownership to. Note that you must be owner of the server to do this.
- **verification_level** (*VerificationLevel*) – The new verification level for the server.

Raises

- *Forbidden* – You do not have permissions to edit the server.
- *NotFound* – The server you are trying to edit does not exist.
- *HTTPException* – Editing the server failed.
- *InvalidArgument* – The image format passed in to `icon` is invalid. It must be PNG or JPG. This is also raised if you are not the owner of the server and request an ownership transfer.

get_bans(*server*)

This function is a *coroutine*.

Retrieves all the *User* s that are banned from the specified server.

You must have proper permissions to get this information.

Parameters **server** (*Server*) – The server to get ban information from.

Raises

- *Forbidden* – You do not have proper permissions to get the information.
- *HTTPException* – An error occurred while fetching the information.

Returns A list of *User* that have been banned.

Return type list

prune_members(*server*, *, *days*)

This function is a *coroutine*.

Prunes a *Server* from its inactive members.

The inactive members are denoted if they have not logged on in `days` number of days and they have no roles.

You must have the "Kick Members" permission to use this.

To check how many members you would prune without actually pruning, see the *estimate_pruned_members()* function.

Parameters

- **server** (*Server*) – The server to prune from.
- **days** (*int*) – The number of days before counting as inactive.

Raises

- *Forbidden* – You do not have permissions to prune members.
- *HTTPException* – An error occurred while pruning members.
- *InvalidArgument* – An integer was not passed for `days`.

Returns The number of members pruned.

Return type int

estimate_pruned_members(*server*, ***, *days*)
This function is a *coroutine*.

Similar to *prune_members()* except instead of actually pruning members, it returns how many members it would prune from the server had it been called.

Parameters

- **server** (*Server*) – The server to estimate a prune from.
- **days** (*int*) – The number of days before counting as inactive.

Raises

- *Forbidden* – You do not have permissions to prune members.
- *HTTPException* – An error occurred while fetching the prune members estimate.
- *InvalidArgument* – An integer was not passed for `days`.

Returns The number of members estimated to be pruned.

Return type int

create_custom_emoji(*server*, ***, *name*, *image*)
This function is a *coroutine*.

Creates a custom *Emoji* for a *Server*.

This endpoint is only allowed for user bots or white listed bots. If this is done by a user bot then this is a local emoji that can only be used inside that server.

There is currently a limit of 50 local emotes per server.

Parameters

- **server** (*Server*) – The server to add the emoji to.
- **name** (*str*) – The emoji name. Must be at least 2 characters.
- **image** (*bytes*) – The *bytes-like* object representing the image data to use. Only JPG and PNG images are supported.

Returns The created emoji.

Return type *Emoji*

Raises

- *Forbidden* – You are not allowed to create emojis.
- *HTTPException* – An error occurred creating an emoji.

delete_custom_emoji(*emoji*)
This function is a *coroutine*.

Deletes a custom *Emoji* from a *Server*.

This follows the same rules as *create_custom_emoji()*.

Parameters **emoji** (*Emoji*) – The emoji to delete.

Raises

- *Forbidden* – You are not allowed to delete emojis.
- *HTTPException* – An error occurred deleting the emoji.

edit_custom_emoji(*emoji*, *, *name*)
This function is a *coroutine*.

Edits a *Emoji*.

Parameters

- **emoji** (*Emoji*) – The emoji to edit.
- **name** (*str*) – The new emoji name.

Raises

- *Forbidden* – You are not allowed to edit emojis.
- *HTTPException* – An error occurred editing the emoji.

create_invite(*destination*, **options*)
This function is a *coroutine*.

Creates an invite for the destination which could be either a *Server* or *Channel*.

Parameters

- **destination** – The *Server* or *Channel* to create the invite to.
- **max_age** (*int*) – How long the invite should last. If it's 0 then the invite doesn't expire. Defaults to 0.
- **max_uses** (*int*) – How many uses the invite could be used for. If it's 0 then there are unlimited uses. Defaults to 0.
- **temporary** (*bool*) – Denotes that the invite grants temporary membership (i.e. they get kicked after they disconnect). Defaults to False.
- **unique** (*bool*) – Indicates if a unique invite URL should be created. Defaults to True. If this is set to False then it will return a previously created invite.

Raises *HTTPException* – Invite creation failed.

Returns The invite that was created.

Return type *Invite*

get_invite(*url*)
This function is a *coroutine*.

Gets a *Invite* from a discord.gg URL or ID.

Note: If the invite is for a server you have not joined, the server and channel attributes of the returned invite will be *Object* with the names patched in.

Parameters **url** (*str*) – The discord invite ID or URL (must be a discord.gg URL).

Raises

- *NotFound* – The invite has expired or is invalid.
- *HTTPException* – Getting the invite failed.

Returns The invite from the URL/ID.

Return type *Invite*

invites_from(*server*)
This function is a *coroutine*.

Returns a list of all active instant invites from a *Server*.

You must have proper permissions to get this information.

Parameters **server** (*Server*) – The server to get invites from.

Raises

- *Forbidden* – You do not have proper permissions to get the information.
- *HTTPException* – An error occurred while fetching the information.

Returns The list of invites that are currently active.

Return type list of *Invite*

accept_invite(*invite*)
This function is a *coroutine*.

Accepts an *Invite*, URL or ID to an invite.

The URL must be a discord.gg URL. e.g. "http://discord.gg/codehere". An ID for the invite is just the "codehere" portion of the invite URL.

Parameters **invite** – The *Invite* or URL to an invite to accept.

Raises

- *HTTPException* – Accepting the invite failed.
- *NotFound* – The invite is invalid or expired.
- *Forbidden* – You are a bot user and cannot use this endpoint.

delete_invite(*invite*)
This function is a *coroutine*.

Revokes an *Invite*, URL, or ID to an invite.

The `invite` parameter follows the same rules as *accept_invite()*.

Parameters **invite** – The invite to revoke.

Raises

- *Forbidden* – You do not have permissions to revoke invites.
- *NotFound* – The invite is invalid or expired.
- *HTTPException* – Revoking the invite failed.

move_role(*server*, *role*, *position*)
This function is a *coroutine*.

Moves the specified *Role* to the given position in the *Server*.

The *Role* object is not directly modified afterwards until the corresponding WebSocket event is received.

Parameters

- **server** (*Server*) – The server the role belongs to.
- **role** (*Role*) – The role to edit.
- **position** (*int*) – The position to insert the role to.

Raises

- *InvalidArgument* – If position is 0, or role is server.default_role
- *Forbidden* – You do not have permissions to change role order.
- *HTTPException* – If moving the role failed, or you are of too low rank to move the role.

edit_role(*server*, *role*, ***fields*)
This function is a *coroutine*.

Edits the specified *Role* for the entire *Server*.

The *Role* object is not directly modified afterwards until the corresponding WebSocket event is received.

All fields except `server` and `role` are optional. To change the position of a role, use *move_role()* instead.

Changed in version 0.8.0: Editing now uses keyword arguments instead of editing the *Role* object directly.

Parameters

- **server** (*Server*) – The server the role belongs to.
- **role** (*Role*) – The role to edit.
- **name** (*str*) – The new role name to change to.
- **permissions** (*Permissions*) – The new permissions to change to.
- **colour** (*Colour*) – The new colour to change to. (aliased to color as well)
- **hoist** (*bool*) – Indicates if the role should be shown separately in the online list.
- **mentionable** (*bool*) – Indicates if the role should be mentionable by others.

Raises

- *Forbidden* – You do not have permissions to change the role.
- *HTTPException* – Editing the role failed.

delete_role(*server*, *role*)
This function is a *coroutine*.

Deletes the specified *Role* for the entire *Server*.

Parameters

- **server** (*Server*) – The server the role belongs to.
- **role** (*Role*) – The role to delete.

Raises

- *Forbidden* – You do not have permissions to delete the role.
- *HTTPException* – Deleting the role failed.

add_roles(*member*, **roles*)
This function is a *coroutine*.

Gives the specified *Member* a number of *Role* s.

You must have the proper permissions to use this function.

The *Member* object is not directly modified afterwards until the corresponding WebSocket event is received.

Parameters

- **member** (*Member*) – The member to give roles to.
- ***roles** – An argument list of *Role* s to give the member.

Raises

- *Forbidden* – You do not have permissions to add roles.
- *HTTPException* – Adding roles failed.

remove_roles(*member*, **roles*)
This function is a *coroutine*.

Removes the *Role* s from the *Member*.

You must have the proper permissions to use this function.

The *Member* object is not directly modified afterwards until the corresponding WebSocket event is received.

Parameters

- **member** (*Member*) – The member to revoke roles from.
- ***roles** – An argument list of *Role* s to revoke the member.

Raises

- *Forbidden* – You do not have permissions to revoke roles.
- *HTTPException* – Removing roles failed.

replace_roles(*member*, **roles*)
This function is a *coroutine*.

Replaces the *Member*'s roles.

You must have the proper permissions to use this function.

This function **replaces** all roles that the member has. For example if the member has roles `[a, b, c]` and the call is `client.replace_roles(member, d, e, c)` then the member has the roles `[d, e, c]`.

The *Member* object is not directly modified afterwards until the corresponding WebSocket event is received.

Parameters

- **member** (*Member*) – The member to replace roles from.
- ***roles** – An argument list of *Role* s to replace the roles with.

Raises

- *Forbidden* – You do not have permissions to revoke roles.
- *HTTPException* – Removing roles failed.

create_role(*server*, ***fields*)
This function is a *coroutine*.

Creates a *Role*.

This function is similar to *edit_role* in both the fields taken and exceptions thrown.

Returns The newly created role. This not the same role that is stored in cache.

Return type *Role*

edit_channel_permissions(*channel*, *target*, *overwrite=None*)
This function is a *coroutine*.

Sets the channel specific permission overwrites for a target in the specified *Channel*.

The `target` parameter should either be a *Member* or a *Role* that belongs to the channel's server.

You must have the proper permissions to do this.

Examples

Setting allow and deny:

```
overwrite = discord.PermissionOverwrite()
overwrite.read_messages = True
overwrite.ban_members = False
await client.edit_channel_permissions(message.channel, message.author,␣
↪overwrite)
```

Parameters

- **channel** (*Channel*) – The channel to give the specific permissions for.
- **target** – The *Member* or *Role* to overwrite permissions for.
- **overwrite** (*PermissionOverwrite*) – The permissions to allow and deny to the target.

Raises

- *Forbidden* – You do not have permissions to edit channel specific permissions.
- *NotFound* – The channel specified was not found.
- *HTTPException* – Editing channel specific permissions failed.
- *InvalidArgument* – The overwrite parameter was not of type *PermissionOverwrite* or the target type was not *Role* or *Member*.

delete_channel_permissions(*channel*, *target*)
This function is a *coroutine*.

Removes a channel specific permission overwrites for a target in the specified *Channel*.

The target parameter follows the same rules as *edit_channel_permissions()*.

You must have the proper permissions to do this.

Parameters

- **channel** (*Channel*) – The channel to give the specific permissions for.
- **target** – The *Member* or *Role* to overwrite permissions for.

Raises

- *Forbidden* – You do not have permissions to delete channel specific permissions.
- *NotFound* – The channel specified was not found.
- *HTTPException* – Deleting channel specific permissions failed.

move_member(*member*, *channel*)
This function is a *coroutine*.

Moves a *Member* to a different voice channel.

You must have proper permissions to do this.

Note: You cannot pass in a *Object* instead of a *Channel* object in this function.

Parameters

- **member** (*Member*) – The member to move to another voice channel.
- **channel** (*Channel*) – The voice channel to move the member to.

Raises

- *InvalidArgument* – The channel provided is not a voice channel.
- *HTTPException* – Moving the member failed.
- *Forbidden* – You do not have permissions to move the member.

join_voice_channel(*channel*)
This function is a *coroutine*.

Joins a voice channel and creates a *VoiceClient* to establish your connection to the voice server.

After this function is successfully called, `voice` is set to the returned *VoiceClient*.

Parameters **channel** (*Channel*) – The voice channel to join to.

Raises

- *InvalidArgument* – The channel was not a voice channel.
- `asyncio.TimeoutError` – Could not connect to the voice channel in time.
- *ClientException* – You are already connected to a voice channel.
- `OpusNotLoaded` – The opus library has not been loaded.

Returns A voice client that is fully connected to the voice server.

Return type *VoiceClient*

is_voice_connected(*server*)
Indicates if we are currently connected to a voice channel in the specified server.

Parameters **server** (*Server*) – The server to query if we're connected to it.

voice_client_in(*server*)
Returns the voice client associated with a server.

If no voice client is found then `None` is returned.

Parameters **server** (*Server*) – The server to query if we have a voice client for.

Returns The voice client associated with the server.

Return type *VoiceClient*

group_call_in(*channel*)
Returns the *GroupCall* associated with a private channel.

If no group call is found then `None` is returned.

Parameters **channel** (*PrivateChannel*) – The group private channel to query the group call for.

Returns The group call.

Return type Optional[*GroupCall*]

application_info()
This function is a *coroutine*.

Retrieve's the bot's application information.

Returns A namedtuple representing the application info.

Return type *AppInfo*

Raises *HTTPException* – Retrieving the information failed somehow.

get_user_info(*user_id*)
This function is a *coroutine*.

Retrieves a *User* based on their ID. This can only be used by bot accounts. You do not have to share any servers with the user to get this information, however many operations do require that you do.

Parameters **user_id** (*str*) – The user's ID to fetch from.

Returns The user you requested.

Return type *User*

Raises

- *NotFound* – A user with this ID does not exist.
- *HTTPException* – Fetching the user failed.

4.3 Voice

class discord.**VoiceClient**(*user*, *main_ws*, *session_id*, *channel*, *data*, *loop*)
Represents a Discord voice connection.

This client is created solely through *Client.join_voice_channel()* and its only purpose is to transmit voice.

> **Warning:** In order to play audio, you must have loaded the opus library through *opus.load_opus()*.
>
> If you don't do this then the library will not be able to transmit audio.

session_id
str – The voice connection session ID.

token
str – The voice connection token.

user
User – The user connected to voice.

endpoint
str – The endpoint we are connecting to.

channel
Channel – The voice channel connected to.

server
Server – The server the voice channel is connected to. Shorthand for `channel.server`.

loop
The event loop that the voice client is running on.

poll_voice_ws()
This function is a *coroutine*. Reads from the voice websocket while connected.

disconnect()
This function is a *coroutine*.

Disconnects all connections to the voice client.

In order to reconnect, you must create another voice client using *Client.join_voice_channel()*.

move_to(*channel*)
This function is a *coroutine*.

Moves you to a different voice channel.

> **Warning:** *Object* instances do not work with this function.

Parameters **channel** (*Channel*) – The channel to move to. Must be a voice channel.

Raises *InvalidArgument* – Not a voice channel.

is_connected()
bool : Indicates if the voice client is connected to voice.

create_ffmpeg_player(*filename*, *, *use_avconv=False*, *pipe=False*, *stderr=None*, *options=None*, *before_options=None*, *headers=None*, *after=None*)
Creates a stream player for ffmpeg that launches in a separate thread to play audio.

The ffmpeg player launches a subprocess of `ffmpeg` to a specific filename and then plays that file.

You must have the ffmpeg or avconv executable in your path environment variable in order for this to work.

The operations that can be done on the player are the same as those in *create_stream_player()*.

Examples

Basic usage:

```
voice = await client.join_voice_channel(channel)
player = voice.create_ffmpeg_player('cool.mp3')
player.start()
```

Parameters

- **filename** – The filename that ffmpeg will take and convert to PCM bytes. If `pipe` is True then this is a file-like object that is passed to the stdin of `ffmpeg`.
- **use_avconv** (*bool*) – Use `avconv` instead of `ffmpeg`.
- **pipe** (*bool*) – If true, denotes that `filename` parameter will be passed to the stdin of ffmpeg.
- **stderr** – A file-like object or `subprocess.PIPE` to pass to the Popen constructor.
- **options** (*str*) – Extra command line flags to pass to `ffmpeg` after the `-i` flag.
- **before_options** (*str*) – Command line flags to pass to `ffmpeg` before the `-i` flag.
- **headers** (*dict*) – HTTP headers dictionary to pass to `-headers` command line option
- **after** (*callable*) – The finalizer that is called after the stream is done being played. All exceptions the finalizer throws are silently discarded.

Raises *ClientException* – Popen failed to due to an error in `ffmpeg` or `avconv`.

Returns A stream player with specific operations. See *create_stream_player()*.

Return type StreamPlayer

create_ytdl_player(*url*, *, *ytdl_options=None*, ***kwargs*)

This function is a *coroutine*.

Creates a stream player for youtube or other services that launches in a separate thread to play the audio.

The player uses the `youtube_dl` python library to get the information required to get audio from the URL. Since this uses an external library, you must install it yourself. You can do so by calling `pip install youtube_dl`.

You must have the ffmpeg or avconv executable in your path environment variable in order for this to work.

The operations that can be done on the player are the same as those in *create_stream_player()*. The player has been augmented and enhanced to have some info extracted from the URL. If youtube-dl fails to extract the information then the attribute is `None`. The `yt`, `url`, and `download_url` attributes are always available.

Operation	Description
player.yt	The *YoutubeDL <ytdl>* instance.
player.url	The URL that is currently playing.
player.download_url	The URL that is currently being downloaded to ffmpeg.
player.title	The title of the audio stream.
player.description	The description of the audio stream.
player.uploader	The uploader of the audio stream.
player.upload_date	A datetime.date object of when the stream was uploaded.
player.duration	The duration of the audio in seconds.
player.likes	How many likes the audio stream has.
player.dislikes	How many dislikes the audio stream has.
player.is_live	Checks if the audio stream is currently livestreaming.
player.views	How many views the audio stream has.

Examples

Basic usage:

```
voice = await client.join_voice_channel(channel)
player = await voice.create_ytdl_player('https://www.youtube.com/watch?
↪v=d62TYemN6MQ')
player.start()
```

Parameters

- **url** (*str*) – The URL that `youtube_dl` will take and download audio to pass to `ffmpeg` or `avconv` to convert to PCM bytes.
- **ytdl_options** (*dict*) – A dictionary of options to pass into the `YoutubeDL` instance. See the documentation for more details.
- ****kwargs** – The rest of the keyword arguments are forwarded to *create_ffmpeg_player()*.

Raises *ClientException* – Popen failure from either `ffmpeg`/`avconv`.

Returns An augmented StreamPlayer that uses ffmpeg. See *create_stream_player()* for base operations.

Return type StreamPlayer

encoder_options(*, *sample_rate*, *channels=2*)

Sets the encoder options for the OpusEncoder.

Calling this after you create a stream player via *create_ffmpeg_player()* or *create_stream_player()* has no effect.

Parameters

- **sample_rate** (*int*) – Sets the sample rate of the OpusEncoder. The unit is in Hz.
- **channels** (*int*) – Sets the number of channels for the OpusEncoder. 2 for stereo, 1 for mono.

Raises *InvalidArgument* – The values provided are invalid.

create_stream_player(*stream*, *, *after=None*)

Creates a stream player that launches in a separate thread to play audio.

The stream player assumes that `stream.read` is a valid function that returns a *bytes-like* object.

The finalizer, `after` is called after the stream has been exhausted or an error occurred (see below).

The following operations are valid on the `StreamPlayer` object:

Operation	Description
player.start()	Starts the audio stream.
player.stop()	Stops the audio stream.
player.is_done()	Returns a bool indicating if the stream is done.
player.is_playing()	Returns a bool indicating if the stream is playing.
player.pause()	Pauses the audio stream.
player.resume()	Resumes the audio stream.
player.volume	Allows you to set the volume of the stream. 1.0 is equivalent to 100% and 0.0 is equal to 0%. The maximum the volume can be set to is 2.0 for 200%.
player.error	The exception that stopped the player. If no error happened, then this returns None.

The stream must have the same sampling rate as the encoder and the same number of channels. The defaults are 48000 Hz and 2 channels. You could change the encoder options by using *encoder_options()* but this must be called **before** this function.

If an error happens while the player is running, the exception is caught and the player is then stopped. The caught exception could then be retrieved via `player.error`. When the player is stopped in this matter, the finalizer under `after` is called.

Parameters

- **stream** – The stream object to read from.
- **after** – The finalizer that is called after the stream is exhausted. All exceptions it throws are silently discarded. This function can have either no parameters or a single parameter taking in the current player.

Returns A stream player with the operations noted above.

Return type StreamPlayer

play_audio(*data*, *, *encode=True*)

Sends an audio packet composed of the data.

You must be connected to play audio.

Parameters

- **data** (*bytes*) – The *bytes-like object* denoting PCM or Opus voice data.
- **encode** (*bool*) – Indicates if `data` should be encoded into Opus.

Raises

- *ClientException* – You are not connected.
- `OpusError` – Encoding the data failed.

4.3.1 Opus Library

discord.opus.**load_opus**(*name*)

Loads the libopus shared library for use with voice.

If this function is not called then the library uses the function ctypes.util.find_library and then loads that one if available.

Not loading a library leads to voice not working.

This function propagates the exceptions thrown.

Warning: The bitness of the library must match the bitness of your python interpreter. If the library is 64-bit then your python interpreter must be 64-bit as well. Usually if there's a mismatch in bitness then the load will throw an exception.

Note: On Windows, the .dll extension is not necessary. However, on Linux the full extension is required to load the library, e.g. `libopus.so.1`. On Linux however, find library will usually find the library automatically without you having to call this.

Parameters **name** (*str*) – The filename of the shared library.

discord.opus.**is_loaded**()
Function to check if opus lib is successfully loaded either via the ctypes.util.find_library call of *load_opus()*.

This must return True for voice to work.

Returns Indicates if the opus library has been loaded.

Return type bool

4.4 Event Reference

This page outlines the different types of events listened by *Client*.

There are two ways to register an event, the first way is through the use of *Client.event()*. The second way is through subclassing *Client* and overriding the specific events. For example:

```
import discord

class MyClient(discord.Client):

    @asyncio.coroutine
    def on_message(self, message):
        yield from self.send_message(message.channel, 'Hello World!')
```

If an event handler raises an exception, *on_error()* will be called to handle it, which defaults to print a traceback and ignore the exception.

Warning: All the events must be a *coroutine*. If they aren't, then you might get unexpected errors. In order to turn a function into a coroutine they must either be decorated with @asyncio.coroutine or in Python 3.5+ be defined using the async def declaration.

The following two functions are examples of coroutine functions:

```
async def on_ready():
    pass

@asyncio.coroutine
def on_ready():
    pass
```

Since this can be a potentially common mistake, there is a helper decorator, *Client.async_event()* to convert a basic function into a coroutine and an event at the same time. Note that it is not necessary if you use async def.

New in version 0.7.0: Subclassing to listen to events.

discord.**on_ready**()
Called when the client is done preparing the data received from Discord. Usually after login is successful and the *Client.servers* and co. are filled up.

Warning: This function is not guaranteed to be the first event called. Likewise, this function is **not** guaranteed to only be called once. This library implements reconnection logic and thus will end up calling this event whenever a RESUME request fails.

discord.**on_resumed**()
Called when the client has resumed a session.

discord.**on_error**(*event*, **args*, ***kwargs*)
Usually when an event raises an uncaught exception, a traceback is printed to stderr and the exception is ignored. If you want to change this behaviour and handle the exception for whatever reason yourself, this event can be overridden. Which, when done, will supress the default action of printing the traceback.

The information of the exception rasied and the exception itself can be retreived with a standard call to `sys.exc_info()`.

If you want exception to propogate out of the *Client* class you can define an `on_error` handler consisting of a single empty `raise` statement. Exceptions raised by `on_error` will not be handled in any way by *Client*.

Parameters

- **event** – The name of the event that raised the exception.
- **args** – The positional arguments for the event that raised the exception.
- **kwargs** – The keyword arguments for the event that raised the execption.

discord.**on_message**(*message*)
Called when a message is created and sent to a server.

Parameters **message** – A *Message* of the current message.

discord.**on_socket_raw_receive**(*msg*)
Called whenever a message is received from the websocket, before it's processed.This event is always dispatched when a message is received and the passed data is not processed in any way.

This is only really useful for grabbing the websocket stream and debugging purposes.

Note: This is only for the messages received from the client websocket. The voice websocket will not trigger this event.

Parameters **msg** – The message passed in from the websocket library. Could be `bytes` for a binary message or `str` for a regular message.

discord.**on_socket_raw_send**(*payload*)
Called whenever a send operation is done on the websocket before the message is sent. The passed parameter is the message that is to sent to the websocket.

This is only really useful for grabbing the websocket stream and debugging purposes.

Note: This is only for the messages received from the client websocket. The voice websocket will not trigger this event.

Parameters **payload** – The message that is about to be passed on to the websocket library. It can be `bytes` to denote a binary message or `str` to denote a regular text message.

discord.**on_message_delete**(*message*)
Called when a message is deleted. If the message is not found in the *Client.messages* cache, then these events will not be called. This happens if the message is too old or the client is participating in high traffic servers. To fix this, increase the `max_messages` option of *Client*.

Parameters **message** – A *Message* of the deleted message.

discord.**on_message_edit**(*before*, *after*)
Called when a message receives an update event. If the message is not found in the *Client.messages* cache, then these events will not be called. This happens if the message is too old or the client is participating in high traffic servers. To fix this, increase the max_messages option of *Client*.

The following non-exhaustive cases trigger this event:

- A message has been pinned or unpinned.
- The message content has been changed.
- **The message has received an embed.**
  - For performance reasons, the embed server does not do this in a "consistent" manner.
- A call message has received an update to its participants or ending time.

Parameters

- **before** – A *Message* of the previous version of the message.
- **after** – A *Message* of the current version of the message.

discord.**on_reaction_add**(*reaction*, *user*)
Called when a message has a reaction added to it. Similar to on_message_edit, if the message is not found in the *Client.messages* cache, then this event will not be called.

Note: To get the message being reacted, access it via *Reaction.message*.

Parameters

- **reaction** – A *Reaction* showing the current state of the reaction.
- **user** – A *User* or *Member* of the user who added the reaction.

discord.**on_reaction_remove**(*reaction*, *user*)
Called when a message has a reaction removed from it. Similar to on_message_edit, if the message is not found in the *Client.messages* cache, then this event will not be called.

Note: To get the message being reacted, access it via *Reaction.message*.

Parameters

- **reaction** – A *Reaction* showing the current state of the reaction.
- **user** – A *User* or *Member* of the user who removed the reaction.

discord.**on_reaction_clear**(*message*, *reactions*)
Called when a message has all its reactions removed from it. Similar to on_message_edit, if the message is not found in the *Client.messages* cache, then this event will not be called.

Parameters

- **message** – The *Message* that had its reactions cleared.
- **reactions** – A list of *Reaction*s that were removed.

discord.**on_channel_delete**(*channel*)

discord.**on_channel_create**(*channel*)
Called whenever a channel is removed or added from a server.

Note that you can get the server from *Channel.server*. *on_channel_create()* could also pass in a *PrivateChannel* depending on the value of *Channel.is_private*.

Parameters **channel** – The *Channel* that got added or deleted.

discord.**on_channel_update**(*before*, *after*)
Called whenever a channel is updated. e.g. changed name, topic, permissions.

Parameters

- **before** – The *Channel* that got updated with the old info.
- **after** – The *Channel* that got updated with the updated info.

discord.**on_member_join**(*member*)
discord.**on_member_remove**(*member*)
Called when a *Member* leaves or joins a *Server*.

Parameters **member** – The *Member* that joined or left.

discord.**on_member_update**(*before*, *after*)
Called when a *Member* updates their profile.

This is called when one or more of the following things change:

- status
- game playing
- avatar
- nickname
- roles

Parameters

- **before** – The *Member* that updated their profile with the old info.
- **after** – The *Member* that updated their profile with the updated info.

discord.**on_server_join**(*server*)
Called when a *Server* is either created by the *Client* or when the *Client* joins a server.

Parameters **server** – The class:*Server* that was joined.

discord.**on_server_remove**(*server*)
Called when a *Server* is removed from the *Client*.

This happens through, but not limited to, these circumstances:

- The client got banned.
- The client got kicked.
- The client left the server.
- The client or the server owner deleted the server.

In order for this event to be invoked then the *Client* must have been part of the server to begin with. (i.e. it is part of *Client.servers*)

Parameters **server** – The *Server* that got removed.

discord.**on_server_update**(*before*, *after*)
Called when a *Server* updates, for example:

- Changed name
- Changed AFK channel
- Changed AFK timeout
- etc

Parameters

- **before** – The *Server* prior to being updated.
- **after** – The *Server* after being updated.

discord.**on_server_role_create**(*role*)
discord.**on_server_role_delete**(*role*)
Called when a *Server* creates or deletes a new *Role*.

To get the server it belongs to, use *Role.server*.

Parameters **role** – The *Role* that was created or deleted.

discord.**on_server_role_update**(*before*, *after*)
Called when a *Role* is changed server-wide.

Parameters

- **before** – The *Role* that updated with the old info.
- **after** – The *Role* that updated with the updated info.

discord.**on_server_emojis_update**(*before*, *after*)
Called when a *Server* adds or removes *Emoji*.

Parameters

- **before** – A list of *Emoji* before the update.
- **after** – A list of *Emoji* after the update.

discord.**on_server_available**(*server*)
discord.**on_server_unavailable**(*server*)
Called when a server becomes available or unavailable. The server must have existed in the *Client.servers* cache.

Parameters **server** – The *Server* that has changed availability.

discord.**on_voice_state_update**(*before*, *after*)
Called when a *Member* changes their voice state.

The following, but not limited to, examples illustrate when this event is called:

- A member joins a voice room.
- A member leaves a voice room.
- A member is muted or deafened by their own accord.
- A member is muted or deafened by a server administrator.

Parameters

- **before** – The *Member* whose voice state changed prior to the changes.

- **after** – The *Member* whose voice state changed after the changes.

discord.**on_member_ban**(*member*)
Called when a *Member* gets banned from a *Server*.

You can access the server that the member got banned from via *Member.server*.

Parameters **member** – The member that got banned.

discord.**on_member_unban**(*server*, *user*)
Called when a *User* gets unbanned from a *Server*.

Parameters

- **server** – The server the user got unbanned from.
- **user** – The user that got unbanned.

discord.**on_typing**(*channel*, *user*, *when*)
Called when someone begins typing a message.

The `channel` parameter could either be a *PrivateChannel* or a *Channel*. If `channel` is a *PrivateChannel* then the `user` parameter is a *User*, otherwise it is a *Member*.

Parameters

- **channel** – The location where the typing originated from.
- **user** – The user that started typing.
- **when** – A `datetime.datetime` object representing when typing started.

discord.**on_group_join**(*channel*, *user*)
discord.**on_group_remove**(*channel*, *user*)
Called when someone joins or leaves a group, i.e. a *PrivateChannel* with a *PrivateChannel.type* of *ChannelType.group*.

Parameters

- **channel** – The group that the user joined or left.
- **user** – The user that joined or left.

4.5 Utility Functions

discord.utils.**find**(*predicate*, *seq*)
A helper to return the first element found in the sequence that meets the predicate. For example:

```
member = find(lambda m: m.name == 'Mighty', channel.server.members)
```

would find the first `Member` whose name is 'Mighty' and return it. If an entry is not found, then `None` is returned.

This is different from filter due to the fact it stops the moment it finds a valid entry.

Parameters

- **predicate** – A function that returns a boolean-like result.
- **seq** (*iterable*) – The iterable to search through.

discord.utils.**get**(*iterable*, ***attrs*)
A helper that returns the first element in the iterable that meets all the traits passed in `attrs`. This is an alternative for *discord.utils.find()*.

When multiple attributes are specified, they are checked using logical AND, not logical OR. Meaning they have to meet every attribute passed in and not one of them.

To have a nested attribute search (i.e. search by `x.y`) then pass in `x__y` as the keyword argument.

If nothing is found that matches the attributes passed, then `None` is returned.

Examples

Basic usage:

```
member = discord.utils.get(message.server.members, name='Foo')
```

Multiple attribute matching:

```
channel = discord.utils.get(server.channels, name='Foo', type=ChannelType.voice)
```

Nested attribute matching:

```
channel = discord.utils.get(client.get_all_channels(), server__name='Cool', name=
↪'general')
```

Parameters

- **iterable** – An iterable to search through.
- ****attrs** – Keyword arguments that denote attributes to search with.

discord.utils.**snowflake_time**(*id*)
Returns the creation date in UTC of a discord id.

discord.utils.**oauth_url**(*client_id*, *permissions=None*, *server=None*, *redirect_uri=None*)
A helper function that returns the OAuth2 URL for inviting the bot into servers.

Parameters

- **client_id** (*str*) – The client ID for your bot.
- **permissions** (`Permissions`) – The permissions you're requesting. If not given then you won't be requesting any permissions.
- **server** (`Server`) – The server to pre-select in the authorization screen, if available.
- **redirect_uri** (*str*) – An optional valid redirect URI.

4.6 Application Info

class discord.**AppInfo**
A namedtuple representing the bot's application info.

id
The application's `client_id`.

name
The application's name.

description
The application's description

icon
The application's icon hash if it exists, `None` otherwise.

icon_url
A property that retrieves the application's icon URL if it exists.

If it doesn't exist an empty string is returned.

owner
The owner of the application. This is a *User* instance with the owner's information at the time of the call.

4.7 Enumerations

The API provides some enumerations for certain types of strings to avoid the API from being stringly typed in case the strings change in the future.

All enumerations are subclasses of enum.

class discord.**ChannelType**
Specifies the type of *Channel*.

text
A text channel.

voice
A voice channel.

private
A private text channel. Also called a direct message.

group
A private group text channel.

category
A server category channel.

class discord.**MessageType**
Specifies the type of *Message*. This is used to denote if a message is to be interpreted as a system message or a regular message.

default
The default message type. This is the same as regular messages.

recipient_add
The system message when a recipient is added to a group private message, i.e. a private channel of type *ChannelType.group*.

recipient_remove
The system message when a recipient is removed from a group private message, i.e. a private channel of type *ChannelType.group*.

call
The system message denoting call state, e.g. missed call, started call, etc.

channel_name_change
The system message denoting that a channel's name has been changed.

channel_icon_change
The system message denoting that a channel's icon has been changed.

pins_add
The system message denoting that a pinned message has been added to a channel.

class discord.**ServerRegion**
Specifies the region a *Server*'s voice server belongs to.

us_west
The US West region.

us_east
The US East region.

us_central
The US Central region.

eu_west
The EU West region.

eu_central
The EU Central region.

singapore
The Singapore region.

london
The London region.

sydney
The Sydney region.

amsterdam
The Amsterdam region.

frankfurt
The Frankfurt region.

brazil
The Brazil region.

vip_us_east
The US East region for VIP servers.

vip_us_west
The US West region for VIP servers.

vip_amsterdam
The Amsterdam region for VIP servers.

class discord.**VerificationLevel**
Specifies a *Server*'s verification level, which is the criteria in which a member must meet before being able to send messages to the server.

none
No criteria set.

low
Member must have a verified email on their Discord account.

medium
Member must have a verified email and be registered on Discord for more than five minutes.

high
Member must have a verified email, be registered on Discord for more than five minutes, and be a member of the server itself for more than ten minutes.

table_flip
An alias for *high*.

class discord.**Status**
Specifies a *Member* 's status.

online
The member is online.

offline
The member is offline.

idle
The member is idle.

dnd
The member is "Do Not Disturb".

do_not_disturb
An alias for *dnd*.

invisible
The member is "invisible". In reality, this is only used in sending a presence a la *Client.change_presence()*. When you receive a user's presence this will be *offline* instead.

4.8 Data Classes

Some classes are just there to be data containers, this lists them.

Note: With the exception of *Object*, *Colour*, and *Permissions* the data classes listed below are **not intended to be created by users** and are also **read-only**.

For example, this means that you should not make your own *User* instances nor should you modify the *User* instance yourself.

If you want to get one of these data classes instances they'd have to be through the cache, and a common way of doing so is through the *utils.find()* function or attributes of data classes that you receive from the events specified in the *Event Reference*.

Warning: Nearly all data classes here have `__slots__` defined which means that it is impossible to have dynamic attributes to the data classes. The only exception to this rule is *Object* which was designed with dynamic attributes in mind.

More information about `__slots__` can be found in the official python documentation.

4.8.1 Object

class discord.**Object**(*id*)
Represents a generic Discord object.

The purpose of this class is to allow you to create 'miniature' versions of data classes if you want to pass in just an ID. Most functions that take in a specific data class with an ID can also take in this class as a substitute instead. Note that even though this is the case, not all objects (if any) actually inherit from this class.

There are also some cases where some websocket events are received in strange order and when such events happened you would receive this class rather than the actual data class. These cases are extremely rare.

id
str – The ID of the object.

created_at
Returns the snowflake's creation time in UTC.

4.8.2 User

class discord.**User**
Represents a Discord user.

Supported Operations:

Operation	Description
x == y	Checks if two users are equal.
x != y	Checks if two users are not equal.
hash(x)	Return the user's hash.
str(x)	Returns the user's name with discriminator.

name
str – The user's username.

id
str – The user's unique ID.

discriminator
str or int – The user's discriminator. This is given when the username has conflicts.

avatar
str – The avatar hash the user has. Could be None.

bot
bool – Specifies if the user is a bot account.

avatar_url
Returns a friendly URL version of the avatar variable the user has. An empty string if the user has no avatar.

default_avatar
Returns the default avatar for a given user. This is calculated by the user's descriminator

default_avatar_url
Returns a URL for a user's default avatar.

mention
Returns a string that allows you to mention the given user.

permissions_in(*channel*)
An alias for *Channel.permissions_for()*.

Basically equivalent to:

```
channel.permissions_for(self)
```

Parameters **channel** – The channel to check your permissions for.

created_at
Returns the user's creation time in UTC.

This is when the user's discord account was created.

display_name
Returns the user's display name.

For regular users this is just their username, but if they have a server specific nickname then that is returned instead.

mentioned_in(*message*)
Checks if the user is mentioned in the specified message.

Parameters **message** (*Message*) – The message to check if you're mentioned in.

4.8.3 Message

class discord.**Message**
Represents a message from Discord.

There should be no need to create one of these manually.

edited_timestamp
Optional[datetime.datetime] – A naive UTC datetime object containing the edited time of the message.

timestamp
datetime.datetime – A naive UTC datetime object containing the time the message was created.

tts
bool – Specifies if the message was done with text-to-speech.

type
MessageType – The type of message. In most cases this should not be checked, but it is helpful in cases where it might be a system message for *system_content*.

author
A *Member* that sent the message. If *channel* is a private channel, then it is a *User* instead.

content
str – The actual contents of the message.

nonce
The value used by the discord server and the client to verify that the message is successfully sent. This is typically non-important.

embeds
list – A list of embedded objects. The elements are objects that meet oEmbed's specification.

channel
The *Channel* that the message was sent from. Could be a *PrivateChannel* if it's a private message. In very rare cases this could be a *Object* instead.

For the sake of convenience, this *Object* instance has an attribute `is_private` set to `True`.

server
Optional[*Server*] – The server that the message belongs to. If not applicable (i.e. a PM) then it's None instead.

call
Optional[*CallMessage*] – The call that the message refers to. This is only applicable to messages of type *MessageType.call*.

mention_everyone
bool – Specifies if the message mentions everyone.

Note: This does not check if the `@everyone` text is in the message itself. Rather this boolean indicates if the `@everyone` text is in the message **and** it did end up mentioning everyone.

---

mentions
list – A list of *Member* that were mentioned. If the message is in a private message then the list will be of *User* instead. For messages that are not of type *MessageType.default*, this array can be used to aid in system messages. For more information, see *system_content*.

Warning: The order of the mentions list is not in any particular order so you should not rely on it. This is a discord limitation, not one with the library.

channel_mentions
list – A list of *Channel* that were mentioned. If the message is in a private message then the list is always empty.

role_mentions
list – A list of *Role* that were mentioned. If the message is in a private message then the list is always empty.

id
str – The message ID.

attachments
list – A list of attachments given to a message.

pinned
bool – Specifies if the message is currently pinned.

reactions
List[*Reaction*] – Reactions to a message. Reactions can be either custom emoji or standard unicode emoji.

raw_mentions
A property that returns an array of user IDs matched with the syntax of <@user_id> in the message content.

This allows you receive the user IDs of mentioned users even in a private message context.

raw_channel_mentions
A property that returns an array of channel IDs matched with the syntax of <#channel_id> in the message content.

raw_role_mentions
A property that returns an array of role IDs matched with the syntax of <@&role_id> in the message content.

clean_content
A property that returns the content in a "cleaned up" manner. This basically means that mentions are transformed into the way the client shows it. e.g. `<#id>` will transform into `#name`.

This will also transform @everyone and @here mentions into non-mentions.

system_content
A property that returns the content that is rendered regardless of the *Message.type*.

In the case of *MessageType.default*, this just returns the regular *Message.content*. Otherwise this returns an English message denoting the contents of the system message.

4.8.4 Reaction

class discord.**Reaction**
Represents a reaction to a message.

Depending on the way this object was created, some of the attributes can have a value of `None`.

Similar to members, the same reaction to a different message are equal.

Supported Operations:

Operation	Description
x == y	Checks if two reactions are the same.
x != y	Checks if two reactions are not the same.
hash(x)	Return the emoji's hash.

emoji
Emoji or str – The reaction emoji. May be a custom emoji, or a unicode emoji.

custom_emoji
bool – If this is a custom emoji.

count
int – Number of times this reaction was made

me
bool – If the user sent this reaction.

message
Message – Message this reaction is for.

4.8.5 Embed

class discord.**Embed**(***kwargs*)
Represents a Discord embed.

The following attributes can be set during creation of the object:

Certain properties return an `EmbedProxy`. Which is a type that acts similar to a regular *dict* except access the attributes via dotted access, e.g. `embed.author.icon_url`. If the attribute is invalid or empty, then a special sentinel value is returned, *Embed.Empty*.

For ease of use, all parameters that expect a `str` are implicitly casted to `str` for you.

title
str – The title of the embed.

type
str – The type of embed. Usually "rich".

description
str – The description of the embed.

url
str – The URL of the embed.

timestamp
datetime.datetime – The timestamp of the embed content.

colour
Colour or int – The colour code of the embed. Aliased to `color` as well.

Empty
A special sentinel value used by `EmbedProxy` and this class to denote that the value or attribute is empty.

footer
Returns a `EmbedProxy` denoting the footer contents.

See *set_footer()* for possible values you can access.

If the attribute has no value then *Empty* is returned.

set_footer(*, *text=Embed.Empty*, *icon_url=Embed.Empty*)
Sets the footer for the embed content.

This function returns the class instance to allow for fluent-style chaining.

Parameters

- **text** (*str*) – The footer text.
- **icon_url** (*str*) – The URL of the footer icon. Only HTTP(S) is supported.

image
Returns a `EmbedProxy` denoting the image contents.

Possible attributes you can access are:

- `url`
- `proxy_url`
- `width`
- `height`

If the attribute has no value then *Empty* is returned.

set_image(*, *url*)
Sets the image for the embed content.

This function returns the class instance to allow for fluent-style chaining.

Parameters **url** (*str*) – The source URL for the image. Only HTTP(S) is supported.

thumbnail
Returns a `EmbedProxy` denoting the thumbnail contents.

Possible attributes you can access are:

- `url`
- `proxy_url`
- `width`

- height

If the attribute has no value then *Empty* is returned.

set_thumbnail(*, *url*)
Sets the thumbnail for the embed content.

This function returns the class instance to allow for fluent-style chaining.

Parameters **url** (*str*) – The source URL for the thumbnail. Only HTTP(S) is supported.

video
Returns a EmbedProxy denoting the video contents.

Possible attributes include:

- url for the video URL.
- height for the video height.
- width for the video width.

If the attribute has no value then *Empty* is returned.

provider
Returns a EmbedProxy denoting the provider contents.

The only attributes that might be accessed are name and url.

If the attribute has no value then *Empty* is returned.

author
Returns a EmbedProxy denoting the author contents.

See *set_author()* for possible values you can access.

If the attribute has no value then *Empty* is returned.

set_author(*, *name*, *url=Embed.Empty*, *icon_url=Embed.Empty*)
Sets the author for the embed content.

This function returns the class instance to allow for fluent-style chaining.

Parameters

- **name** (*str*) – The name of the author.
- **url** (*str*) – The URL for the author.
- **icon_url** (*str*) – The URL of the author icon. Only HTTP(S) is supported.

fields
Returns a list of EmbedProxy denoting the field contents.

See *add_field()* for possible values you can access.

If the attribute has no value then *Empty* is returned.

add_field(*, *name*, *value*, *inline=True*)
Adds a field to the embed object.

This function returns the class instance to allow for fluent-style chaining.

Parameters

- **name** (*str*) – The name of the field.
- **value** (*str*) – The value of the field.

- **inline** (*bool*) – Whether the field should be displayed inline.

clear_fields()
Removes all fields from this embed.

remove_field(*index*)
Removes a field at a specified index.

If the index is invalid or out of bounds then the error is silently swallowed.

Note: When deleting a field by index, the index of the other fields shift to fill the gap just like a regular list.

Parameters **index** (*int*) – The index of the field to remove.

set_field_at(*index*, *, *name*, *value*, *inline=True*)
Modifies a field to the embed object.

The index must point to a valid pre-existing field.

This function returns the class instance to allow for fluent-style chaining.

Parameters

- **index** (*int*) – The index of the field to modify.
- **name** (*str*) – The name of the field.
- **value** (*str*) – The value of the field.
- **inline** (*bool*) – Whether the field should be displayed inline.

Raises IndexError – An invalid index was provided.

to_dict()
Converts this embed object into a dict.

4.8.6 CallMessage

class discord.**CallMessage**
Represents a group call message from Discord.

This is only received in cases where the message type is equivalent to *MessageType.call*.

ended_timestamp
Optional[datetime.datetime] – A naive UTC datetime object that represents the time that the call has ended.

participants
List[*User*] – The list of users that are participating in this call.

message
Message – The message associated with this call message.

call_ended
bool – Indicates if the call has ended.

channel
PrivateChannel– The private channel associated with this message.

duration
Queries the duration of the call.

If the call has not ended then the current duration will be returned.

Returns The timedelta object representing the duration.

Return type datetime.timedelta

4.8.7 GroupCall

class discord.**GroupCall**
Represents the actual group call from Discord.

This is accompanied with a *CallMessage* denoting the information.

call
CallMessage – The call message associated with this group call.

unavailable
bool – Denotes if this group call is unavailable.

ringing
List[*User*] – A list of users that are currently being rung to join the call.

region
ServerRegion – The server region the group call is being hosted on.

connected
A property that returns the list of *User* that are currently in this call.

channel
PrivateChannel– Returns the channel the group call is in.

voice_state_for(*user*)
Retrieves the *VoiceState* for a specified *User*.

If the *User* has no voice state then this function returns `None`.

Parameters **user** (*User*) – The user to retrieve the voice state for.

Returns The voice state associated with this user.

Return type Optiona[*VoiceState*]

4.8.8 Server

class discord.**Server**
Represents a Discord server.

Supported Operations:

Operation	Description
x == y	Checks if two servers are equal.
x != y	Checks if two servers are not equal.
hash(x)	Returns the server's hash.
str(x)	Returns the server's name.

name
str – The server name.

me
Member – Similar to *Client.user* except an instance of *Member*. This is essentially used to get the member version of yourself.

roles
A list of *Role* that the server has available.

emojis
A list of *Emoji* that the server owns.

region
ServerRegion – The region the server belongs on. There is a chance that the region will be a `str` if the value is not recognised by the enumerator.

afk_timeout
int – The timeout to get sent to the AFK channel.

afk_channel
Channel – The channel that denotes the AFK channel. None if it doesn't exist.

members
An iterable of *Member* that are currently on the server.

channels
An iterable of *Channel* that are currently on the server.

icon
str – The server's icon.

id
str – The server's ID.

owner
Member – The member who owns the server.

unavailable
bool – Indicates if the server is unavailable. If this is `True` then the reliability of other attributes outside of *Server.id()* is slim and they might all be None. It is best to not do anything with the server if it is unavailable.

Check the *on_server_unavailable()* and *on_server_available()* events.

large
bool – Indicates if the server is a 'large' server. A large server is defined as having more than `large_threshold` count members, which for this library is set to the maximum of 250.

voice_client
Optional[*VoiceClient*] – The VoiceClient associated with this server. A shortcut for the *Client.voice_client_in()* call.

mfa_level
int – Indicates the server's two factor authorisation level. If this value is 0 then the server does not require 2FA for their administrative members. If the value is 1 then they do.

verification_level
VerificationLevel – The server's verification level.

features
List[str] – A list of features that the server has. They are currently as follows:

- `VIP_REGIONS`: Server has VIP voice regions
- `VANITY_URL`: Server has a vanity invite URL (e.g. discord.gg/discord-api)

- INVITE_SPLASH: Server's invite page has a special splash.

splash
str – The server's invite splash.

get_channel(*channel_id*)
Returns a *Channel* with the given ID. If not found, returns None.

get_member(*user_id*)
Returns a *Member* with the given ID. If not found, returns None.

default_role
Gets the @everyone role that all members have by default.

default_channel
Gets the default *Channel* for the server.

icon_url
Returns the URL version of the server's icon. Returns an empty string if it has no icon.

splash_url
Returns the URL version of the server's invite splash. Returns an empty string if it has no splash.

member_count
Returns the true member count regardless of it being loaded fully or not.

created_at
Returns the server's creation time in UTC.

role_hierarchy
Returns the server's roles in the order of the hierarchy.

The first element of this list will be the highest role in the hierarchy.

get_member_named(*name*)
Returns the first member found that matches the name provided.

The name can have an optional discriminator argument, e.g. "Jake#0001" or "Jake" will both do the lookup. However the former will give a more precise result. Note that the discriminator must have all 4 digits for this to work.

If a nickname is passed, then it is looked up via the nickname. Note however, that a nickname + discriminator combo will not lookup the nickname but rather the username + discriminator combo due to nickname + discriminator not being unique.

If no member is found, None is returned.

Parameters **name** (*str*) – The name of the member to lookup with an optional discriminator.

Returns The member in this server with the associated name. If not found then None is returned.

Return type *Member*

4.8.9 Member

class discord.**Member**
Represents a Discord member to a *Server*.

This is a subclass of *User* that extends more functionality that server members have such as roles and permissions.

voice
VoiceState – The member's voice state. Properties are defined to mirror access of the attributes. e.g. `Member.is_afk` is equivalent to *Member.voice.is_afk*'.

roles
A list of *Role* that the member belongs to. Note that the first element of this list is always the default '@everyone' role.

joined_at
datetime.datetime – A datetime object that specifies the date and time in UTC that the member joined the server for the first time.

status
Status – The member's status. There is a chance that the status will be a `str` if it is a value that is not recognised by the enumerator.

game
Game – The game that the user is currently playing. Could be None if no game is being played.

server
Server – The server that the member belongs to.

nick
Optional[str] – The server specific nickname of the user.

colour
A property that returns a *Colour* denoting the rendered colour for the member. If the default colour is the one rendered then an instance of *Colour.default()* is returned.

There is an alias for this under `color`.

color
A property that returns a *Colour* denoting the rendered colour for the member. If the default colour is the one rendered then an instance of *Colour.default()* is returned.

There is an alias for this under `color`.

top_role
Returns the member's highest role.

This is useful for figuring where a member stands in the role hierarchy chain.

server_permissions
Returns the member's server permissions.

This only takes into consideration the server permissions and not most of the implied permissions or any of the channel permission overwrites. For 100% accurate permission calculation, please use either `permissions_in()` or *Channel.permissions_for()*.

This does take into consideration server ownership and the administrator implication.

4.8.10 VoiceState

class discord.**VoiceState**
Represents a Discord user's voice state.

deaf
bool – Indicates if the user is currently deafened by the server.

mute
bool – Indicates if the user is currently muted by the server.

self_mute
bool – Indicates if the user is currently muted by their own accord.

self_deaf
bool – Indicates if the user is currently deafened by their own accord.

is_afk
bool – Indicates if the user is currently in the AFK channel in the server.

voice_channel
Optional[Union[*Channel*, *PrivateChannel*]] – The voice channel that the user is currently connected to. None if the user is not currently in a voice channel.

4.8.11 Colour

class discord.**Colour**(*value*)
Represents a Discord role colour. This class is similar to an (red, green, blue) tuple.

There is an alias for this called Color.

Supported operations:

Operation	Description
x == y	Checks if two colours are equal.
x != y	Checks if two colours are not equal.
hash(x)	Return the colour's hash.
str(x)	Returns the hex format for the colour.

value
int – The raw integer colour value.

r
Returns the red component of the colour.

g
Returns the green component of the colour.

b
Returns the blue component of the colour.

to_tuple()
Returns an (r, g, b) tuple representing the colour.

classmethod default()
A factory method that returns a *Colour* with a value of 0.

classmethod teal()
A factory method that returns a *Colour* with a value of `0x1abc9c`.

classmethod dark_teal()
A factory method that returns a *Colour* with a value of `0x11806a`.

classmethod green()
A factory method that returns a *Colour* with a value of `0x2ecc71`.

classmethod dark_green()
A factory method that returns a *Colour* with a value of `0x1f8b4c`.

classmethod blue()
A factory method that returns a *Colour* with a value of `0x3498db`.

classmethod dark_blue()
A factory method that returns a *Colour* with a value of 0x206694.

classmethod purple()
A factory method that returns a *Colour* with a value of 0x9b59b6.

classmethod dark_purple()
A factory method that returns a *Colour* with a value of 0x71368a.

classmethod magenta()
A factory method that returns a *Colour* with a value of 0xe91e63.

classmethod dark_magenta()
A factory method that returns a *Colour* with a value of 0xad1457.

classmethod gold()
A factory method that returns a *Colour* with a value of 0xf1c40f.

classmethod dark_gold()
A factory method that returns a *Colour* with a value of 0xc27c0e.

classmethod orange()
A factory method that returns a *Colour* with a value of 0xe67e22.

classmethod dark_orange()
A factory method that returns a *Colour* with a value of 0xa84300.

classmethod red()
A factory method that returns a *Colour* with a value of 0xe74c3c.

classmethod dark_red()
A factory method that returns a *Colour* with a value of 0x992d22.

classmethod lighter_grey()
A factory method that returns a *Colour* with a value of 0x95a5a6.

classmethod dark_grey()
A factory method that returns a *Colour* with a value of 0x607d8b.

classmethod light_grey()
A factory method that returns a *Colour* with a value of 0x979c9f.

classmethod darker_grey()
A factory method that returns a *Colour* with a value of 0x546e7a.

4.8.12 Game

class discord.**Game**(***kwargs*)
Represents a Discord game.

Supported Operations:

Operation	Description
x == y	Checks if two games are equal.
x != y	Checks if two games are not equal.
hash(x)	Return the games's hash.
str(x)	Returns the games's name.

name
str – The game's name.

url
str – The game's URL. Usually used for twitch streaming.

type
int – The type of game being played. 1 indicates "Streaming".

4.8.13 Emoji

class discord.**Emoji**
Represents a custom emoji.

Depending on the way this object was created, some of the attributes can have a value of `None`.

Supported Operations:

Operation	Description
x == y	Checks if two emoji are the same.
x != y	Checks if two emoji are not the same.
hash(x)	Return the emoji's hash.
iter(x)	Returns an iterator of (field, value) pairs. This allows this class to be used as an iterable in list/dict/etc. constructions.
str(x)	Returns the emoji rendered for discord.

name
str – The name of the emoji.

id
str – The emoji's ID.

require_colons
bool – If colons are required to use this emoji in the client (:PJSalt: vs PJSalt).

managed
bool – If this emoji is managed by a Twitch integration.

server
Server – The server the emoji belongs to.

roles
List[*Role*] – A list of *Role* that is allowed to use this emoji. If roles is empty, the emoji is unrestricted.

created_at
Returns the emoji's creation time in UTC.

url
Returns a URL version of the emoji.

4.8.14 Role

class discord.**Role**
Represents a Discord role in a *Server*.

Supported Operations:

Operation	Description
x == y	Checks if two roles are equal.
x != y	Checks if two roles are not equal.
x > y	Checks if a role is higher than another in the hierarchy.
x < y	Checks if a role is lower than another in the hierarchy.
x >= y	Checks if a role is higher or equal to another in the hierarchy.
x <= y	Checks if a role is lower or equal to another in the hierarchy.
hash(x)	Return the role's hash.
str(x)	Returns the role's name.

id
str – The ID for the role.

name
str – The name of the role.

permissions
Permissions – Represents the role's permissions.

server
Server – The server the role belongs to.

colour
Colour – Represents the role colour. An alias exists under `color`.

hoist
bool – Indicates if the role will be displayed separately from other members.

position
int – The position of the role. This number is usually positive. The bottom role has a position of 0.

managed
bool – Indicates if the role is managed by the server through some form of integrations such as Twitch.

mentionable
bool – Indicates if the role can be mentioned by users.

is_everyone
Checks if the role is the @everyone role.

created_at
Returns the role's creation time in UTC.

mention
Returns a string that allows you to mention a role.

4.8.15 Permissions

class discord.**Permissions**(*permissions=0*, ***kwargs*)
Wraps up the Discord permission value.

Supported operations:

Operation	Description
x == y	Checks if two permissions are equal.
x != y	Checks if two permissions are not equal.
x <= y	Checks if a permission is a subset of another permission.
x >= y	Checks if a permission is a superset of another permission.
x < y	Checks if a permission is a strict subset of another permission.
x > y	Checks if a permission is a strict superset of another permission.
hash(x)	Return the permission's hash.
iter(x)	Returns an iterator of (perm, value) pairs. This allows this class to be used as an iterable in e.g. set/list/dict constructions.

The properties provided are two way. You can set and retrieve individual bits using the properties as if they were regular bools. This allows you to edit permissions.

value
The raw value. This value is a bit array field of a 32-bit integer representing the currently available permissions. You should query permissions via the properties rather than using this raw value.

is_subset(*other*)
Returns True if self has the same or fewer permissions as other.

is_superset(*other*)
Returns True if self has the same or more permissions as other.

is_strict_subset(*other*)
Returns True if the permissions on other are a strict subset of those on self.

is_strict_superset(*other*)
Returns True if the permissions on other are a strict superset of those on self.

classmethod none()
A factory method that creates a *Permissions* with all permissions set to False.

classmethod all()
A factory method that creates a *Permissions* with all permissions set to True.

classmethod all_channel()
A *Permissions* with all channel-specific permissions set to True and the server-specific ones set to False. The server-specific permissions are currently:

- manager_server
- kick_members
- ban_members
- administrator
- change_nicknames
- manage_nicknames

classmethod general()
A factory method that creates a *Permissions* with all "General" permissions from the official Discord UI set to True.

classmethod text()
A factory method that creates a *Permissions* with all "Text" permissions from the official Discord UI set to True.

classmethod voice()
A factory method that creates a *Permissions* with all "Voice" permissions from the official Discord UI set to True.

update(***kwargs*)
Bulk updates this permission object.

Allows you to set multiple attributes by using keyword arguments. The names must be equivalent to the properties listed. Extraneous key/value pairs will be silently ignored.

Parameters ****kwargs** – A list of key/value pairs to bulk update permissions with.

create_instant_invite
Returns True if the user can create instant invites.

kick_members
Returns True if the user can kick users from the server.

ban_members
Returns True if a user can ban users from the server.

administrator
Returns True if a user is an administrator. This role overrides all other permissions.

This also bypasses all channel-specific overrides.

manage_channels
Returns True if a user can edit, delete, or create channels in the server.

This also corresponds to the "manage channel" channel-specific override.

manage_server
Returns True if a user can edit server properties.

add_reactions
Returns True if a user can add reactions to messages.

view_audit_logs
Returns True if a user can view the server's audit log.

read_messages
Returns True if a user can read messages from all or specific text channels.

send_messages
Returns True if a user can send messages from all or specific text channels.

send_tts_messages
Returns True if a user can send TTS messages from all or specific text channels.

manage_messages
Returns True if a user can delete messages from a text channel. Note that there are currently no ways to edit other people's messages.

embed_links
Returns True if a user's messages will automatically be embedded by Discord.

attach_files
Returns True if a user can send files in their messages.

read_message_history
Returns True if a user can read a text channel's previous messages.

mention_everyone
Returns True if a user's @everyone will mention everyone in the text channel.

external_emojis
Returns True if a user can use emojis from other servers.

connect
Returns True if a user can connect to a voice channel.

speak
Returns True if a user can speak in a voice channel.

mute_members
Returns True if a user can mute other users.

deafen_members
Returns True if a user can deafen other users.

move_members
Returns True if a user can move users between other voice channels.

use_voice_activation
Returns True if a user can use voice activation in voice channels.

change_nickname
Returns True if a user can change their nickname in the server.

manage_nicknames
Returns True if a user can change other user's nickname in the server.

manage_roles
Returns True if a user can create or edit roles less than their role's position.

This also corresponds to the "manage permissions" channel-specific override.

manage_webhooks
Returns True if a user can create, edit, or delete webhooks.

manage_emojis
Returns True if a user can create, edit, or delete emojis.

4.8.16 PermissionOverwrite

class discord.**PermissionOverwrite**(***kwargs*)
A type that is used to represent a channel specific permission.

Unlike a regular *Permissions*, the default value of a permission is equivalent to `None` and not `False`. Setting a value to `False` is **explicitly** denying that permission, while setting a value to `True` is **explicitly** allowing that permission.

The values supported by this are the same as *Permissions* with the added possibility of it being set to `None`.

Supported operations:

Operation	Description
iter(x)	Returns an iterator of (perm, value) pairs. This allows this class to be used as an iterable in e.g. set/list/dict constructions.

Parameters ****kwargs** – Set the value of permissions by their name.

pair()
Returns the (allow, deny) pair from this overwrite.

The value of these pairs is *Permissions*.

classmethod from_pair(*allow*, *deny*)
Creates an overwrite from an allow/deny pair of *Permissions*.

is_empty()
Checks if the permission overwrite is currently empty.

An empty permission overwrite is one that has no overwrites set to True or False.

update(***kwargs*)
Bulk updates this permission overwrite object.

Allows you to set multiple attributes by using keyword arguments. The names must be equivalent to the properties listed. Extraneous key/value pairs will be silently ignored.

Parameters ****kwargs** – A list of key/value pairs to bulk update with.

4.8.17 Channel

class discord.**Channel**
Represents a Discord server channel.

Supported Operations:

Operation	Description
x == y	Checks if two channels are equal.
x != y	Checks if two channels are not equal.
hash(x)	Returns the channel's hash.
str(x)	Returns the channel's name.

name
str – The channel name.

server
Server – The server the channel belongs to.

id
str – The channel ID.

topic
Optional[str] – The channel's topic. None if it doesn't exist.

is_private
bool – `True` if the channel is a private channel (i.e. PM). `False` in this case.

position
int – The position in the channel list. This is a number that starts at 0. e.g. the top channel is position 0. The position varies depending on being a voice channel or a text channel, so a 0 position voice channel is on top of the voice channel list.

type
ChannelType – The channel type. There is a chance that the type will be `str` if the channel type is not within the ones recognised by the enumerator.

bitrate
int – The channel's preferred audio bitrate in bits per second.

voice_members
A list of Members that are currently inside this voice channel. If *type* is not *ChannelType.voice* then this is always an empty array.

user_limit
int – The channel's limit for number of members that can be in a voice channel.

changed_roles
Returns a list of Roles that have been overridden from their default values in the *Server.roles* attribute.

is_default
bool – Indicates if this is the default channel for the *Server* it belongs to.

mention
str – The string that allows you to mention the channel.

created_at
Returns the channel's creation time in UTC.

overwrites_for(*obj*)
Returns the channel-specific overwrites for a member or a role.

Parameters **obj** – The *Role* or *Member* or *Object* denoting whose overwrite to get.

Returns The permission overwrites for this object.

Return type *PermissionOverwrite*

overwrites
Returns all of the channel's overwrites.

This is returned as a list of two-element tuples containing the target, which can be either a *Role* or a *Member* and the overwrite as the second element as a *PermissionOverwrite*.

Returns The channel's permission overwrites.

Return type List[Tuple[Union[*Role*, *Member*], *PermissionOverwrite*]]

permissions_for(*member*)
Handles permission resolution for the current *Member*.

This function takes into consideration the following cases:

- Server owner
- Server roles
- Channel overrides
- Member overrides
- Whether the channel is the default channel.

Parameters **member** (*Member*) – The member to resolve permissions for.

Returns The resolved permissions for the member.

Return type *Permissions*

4.8.18 PrivateChannel

class discord.**PrivateChannel**
Represents a Discord private channel.

Supported Operations:

Operation	Description
x == y	Checks if two channels are equal.
x != y	Checks if two channels are not equal.
hash(x)	Returns the channel's hash.
str(x)	Returns a string representation of the channel

recipients
list of *User* – The users you are participating with in the private channel.

me
User – The user presenting yourself.

id
str – The private channel ID.

is_private
bool – `True` if the channel is a private channel (i.e. PM). `True` in this case.

type
ChannelType – The type of private channel.

owner
Optional[*User*] – The user that owns the private channel. If the channel type is not *ChannelType.group* then this is always `None`.

icon
Optional[str] – The private channel's icon hash. If the channel type is not *ChannelType.group* then this is always `None`.

name
Optional[str] – The private channel's name. If the channel type is not *ChannelType.group* then this is always `None`.

user
A property that returns the first recipient of the private channel.

This is mainly for compatibility and ease of use with old style private channels that had a single recipient.

icon_url
Returns the channel's icon URL if available or an empty string otherwise.

created_at
Returns the private channel's creation time in UTC.

permissions_for(*user*)
Handles permission resolution for a *User*.

This function is there for compatibility with *Channel*.

Actual private messages do not really have the concept of permissions.

This returns all the Text related permissions set to true except:

- send_tts_messages: You cannot send TTS messages in a PM.
- manage_messages: You cannot delete others messages in a PM.

This also handles permissions for *ChannelType.group* channels such as kicking or mentioning everyone.

Parameters **user** (*User*) – The user to check permissions for.

Returns The resolved permissions for the user.

Return type *Permissions*

4.8.19 Invite

class discord.**Invite**

Represents a Discord *Server* or *Channel* invite.

Depending on the way this object was created, some of the attributes can have a value of `None`.

Supported Operations:

Operation	Description
x == y	Checks if two invites are equal.
x != y	Checks if two invites are not equal.
hash(x)	Return the invite's hash.
str(x)	Returns the invite's URL.

max_age
int – How long the before the invite expires in seconds. A value of 0 indicates that it doesn't expire.

code
str – The URL fragment used for the invite. *xkcd* is also a possible fragment.

server
Server – The server the invite is for.

revoked
bool – Indicates if the invite has been revoked.

created_at
datetime.datetime – A datetime object denoting the time the invite was created.

temporary
bool – Indicates that the invite grants temporary membership. If True, members who joined via this invite will be kicked upon disconnect.

uses
int – How many times the invite has been used.

max_uses
int – How many times the invite can be used.

xkcd
str – The URL fragment used for the invite if it is human readable.

inviter
User – The user who created the invite.

channel
Channel – The channel the invite is for.

id
Returns the proper code portion of the invite.

url
A property that retrieves the invite URL.

4.9 Exceptions

The following exceptions are thrown by the library.

exception discord.**DiscordException**
Base exception class for discord.py

Ideally speaking, this could be caught to handle any exceptions thrown from this library.

exception discord.**ClientException**
Exception that's thrown when an operation in the *Client* fails.

These are usually for exceptions that happened due to user input.

exception discord.**LoginFailure**
Exception that's thrown when the *Client.login()* function fails to log you in from improper credentials or some other misc. failure.

exception discord.**HTTPException**(*response*, *message*)
Exception that's thrown when an HTTP request operation fails.

response
The response of the failed HTTP request. This is an instance of aiohttp.ClientResponse.

text
The text of the error. Could be an empty string.

exception discord.**Forbidden**(*response*, *message*)
Exception that's thrown for when status code 403 occurs.

Subclass of *HTTPException*

exception discord.**NotFound**(*response*, *message*)
Exception that's thrown for when status code 404 occurs.

Subclass of *HTTPException*

exception discord.**InvalidArgument**
Exception that's thrown when an argument to a function is invalid some way (e.g. wrong value or wrong type).

This could be considered the analogous of `ValueError` and `TypeError` except derived from *ClientException* and thus *DiscordException*.

exception discord.**GatewayNotFound**
An exception that is usually thrown when the gateway hub for the *Client* websocket is not found.

exception discord.**ConnectionClosed**(*original*)
Exception that's thrown when the gateway connection is closed for reasons that could not be handled internally.

code
int – The close code of the websocket.

reason
str – The reason provided for the closure.

exception discord.opus.**OpusError**(*code*)
An exception that is thrown for libopus related errors.

code
int – The error code returned.

exception discord.opus.**OpusNotLoaded**
An exception that is thrown for when libopus is not loaded.

CHAPTER 5

Frequently Asked Questions

This is a list of Frequently Asked Questions regarding using `discord.py` and its extension modules. Feel free to suggest a new question or submit one via pull requests.

Questions

5.1 Coroutines

Questions regarding coroutines and asyncio belong here.

5.1.1 I get a SyntaxError around the word `async`! What should I do?

This `SyntaxError` happens because you're using a Python version lower than 3.5. Python 3.4 uses `@asyncio.coroutine` and `yield from` instead of `async def` and `await`.

Thus you must do the following instead:

```
async def foo():
    await bar()

# into

@asyncio.coroutine
def foo():
    yield from bar()
```

Don't forget to `import asyncio` on the top of your files.

It is heavily recommended that you update to Python 3.5 or higher as it simplifies asyncio massively.

5.1.2 What is a coroutine?

A coroutine is a function that must be invoked with `await` or `yield from`. When Python encounters an `await` it stops the function's execution at that point and works on other things until it comes back to that point and finishes off its work. This allows for your program to be doing multiple things at the same time without using threads or complicated multiprocessing.

If you forget to await a coroutine then the coroutine will not run. Never forget to await a coroutine.

5.1.3 Where can I use `await`?

You can only use `await` inside `async def` functions and nowhere else.

5.1.4 What does "blocking" mean?

In asynchronous programming a blocking call is essentially all the parts of the function that are not `await`. Do not despair however, because not all forms of blocking are bad! Using blocking calls is inevitable, but you must work to

make sure that you don't excessively block functions. Remember, if you block for too long then your bot will freeze since it has not stopped the function's execution at that point to do other things.

A common source of blocking for too long is something like `time.sleep(n)`. Don't do that. Use `asyncio.sleep(n)` instead. Similar to this example:

```
# bad
time.sleep(10)

# good
await asyncio.sleep(10)
```

Another common source of blocking for too long is using HTTP requests with the famous module `requests`. While `requests` is an amazing module for non-asynchronous programming, it is not a good choice for `asyncio` because certain requests can block the event loop too long. Instead, use the `aiohttp` library which is installed on the side with this library.

Consider the following example:

```
# bad
r = requests.get('http://random.cat/meow')
if r.status_code == 200:
    js = r.json()
    await client.send_message(channel, js['file'])

# good
async with aiohttp.get('http://random.cat/meow') as r:
    if r.status == 200:
        js = await r.json()
        await client.send_message(channel, js['file'])
```

5.2 General

General questions regarding library usage belong here.

5.2.1 How do I set the "Playing" status?

There is a method for this under *Client* called *Client.change_presence()*. The relevant aspect of this is its `game` keyword argument which takes in a *Game* object. Putting both of these pieces of info together, you get the following:

```
await client.change_presence(game=discord.Game(name='my game'))
```

5.2.2 How do I send a message to a specific channel?

If you have its ID then you can do this in two ways, first is by using *Object*:

```
await client.send_message(discord.Object(id='12324234183172'), 'hello')
```

The second way is by calling *Client.get_channel()* directly:

```
await client.send_message(client.get_channel('12324234183172'), 'hello')
```

5.2.3 I'm passing IDs as integers and things are not working!

In the library IDs must be of type `str` not of type `int`. Wrap it in quotes.

5.2.4 How do I upload an image?

There are two ways of doing it. Both of which involve using *Client.send_file()*.

The first is by opening the file and passing it directly:

```
with open('my_image.png', 'rb') as f:
    await client.send_file(channel, f)
```

The second is by passing the file name directly:

```
await client.send_file(channel, 'my_image.png')
```

5.2.5 How can I add a reaction to a message?

You use the *Client.add_reaction()* method.

If you want to use unicode emoji, you must pass a valid unicode code point in a string. In your code, you can write this in a few different ways:

- `''`
- `'\U0001F44D'`
- `'\N{THUMBS UP SIGN}'`

In case you want to use emoji that come from a message, you already get their code points in the content without needing to do anything special. You **cannot** send `':thumbsup:'` style shorthands.

For custom emoji, you should pass an instance of *discord.Emoji*. You can also pass a `'name:id'` string, but if you can use said emoji, you should be able to use *Client.get_all_emojis()*/*Server.emojis* to find the one you're looking for.

5.2.6 How do I pass a coroutine to the player's "after" function?

A StreamPlayer is just a `threading.Thread` object that plays music. As a result it does not execute inside a coroutine. This does not mean that it is not possible to call a coroutine in the `after` parameter. To do so you must pass a callable that wraps up a couple of aspects.

The first gotcha that you must be aware of is that calling a coroutine is not a thread-safe operation. Since we are technically in another thread, we must take caution in calling thread-safe operations so things do not bug out. Luckily for us, `asyncio` comes with a `asyncio.run_coroutine_threadsafe` function that allows us to call a coroutine from another thread.

> **Warning:** This function is only part of 3.5.1+ and 3.4.4+. If you are not using these Python versions then use `discord.compat.run_coroutine_threadsafe`.

However, this function returns a `concurrent.Future` and to actually call it we have to fetch its result. Putting all of this together we can do the following:

```
def my_after():
    coro = client.send_message(some_channel, 'Song is done!')
    fut = asyncio.run_coroutine_threadsafe(coro, client.loop)
    try:
        fut.result()
    except:
        # an error happened sending the message
        pass

player = await voice.create_ytdl_player(url, after=my_after)
player.start()
```

5.2.7 Why is my "after" function being called right away?

The `after` keyword argument expects a *function object* to be passed in. Similar to how `threading.Thread` expects a callable in its `target` keyword argument. This means that the following are invalid:

```
player = await voice.create_ytdl_player(url, after=self.foo())
other  = await voice.create_ytdl_player(url, after=self.bar(10))
```

However the following are correct:

```
player = await voice.create_ytdl_player(url, after=self.foo)
other  = await voice.create_ytdl_player(url, after=lambda: self.bar(10))
```

Basically, these functions should not be called.

5.2.8 How do I run something in the background?

Check the background_task.py example.

5.2.9 How do I get a specific User/Role/Channel/Server?

There are multiple ways of doing this. If you have a specific entity's ID then you can use one of the following functions:

- *Client.get_channel()*
- *Client.get_server()*
- *Server.get_member()*
- *Server.get_channel()*

If the functions above do not help you, then use of *utils.find()* or *utils.get()* would serve some use in finding specific entities. The documentation for those functions provides specific examples.

5.3 Commands Extension

Questions regarding `discord.ext.commands` belong here.

5.3.1 Is there any documentation for this?

Not at the moment. Writing documentation for stuff takes time. A lot of people get by reading the docstrings in the source code. Others get by via asking questions in the Discord server. Others look at the source code of other existing bots.

There is a basic example showcasing some functionality.

Documentation is being worked on, it will just take some time to polish it.

5.3.2 Why does `on_message` make my commands stop working?

Overriding the default provided `on_message` forbids any extra commands from running. To fix this, add a `bot.process_commands(message)` line at the end of your `on_message`. For example:

```
@bot.event
async def on_message(message):
    # do some extra stuff here

    await bot.process_commands(message)
```

5.3.3 Can I use `bot.say` in other places aside from commands?

No. They only work inside commands due to the way the magic involved works.

5.3.4 Why do my arguments require quotes?

In a simple command defined as:

```
@bot.command()
async def echo(message: str):
    await bot.say(message)
```

Calling it via `?echo a b c` will only fetch the first argument and disregard the rest. To fix this you should either call it via `?echo "a b c"` or change the signature to have "consume rest" behaviour. Example:

```
@bot.command()
async def echo(*, message: str):
    await bot.say(message)
```

This will allow you to use `?echo a b c` without needing the quotes.

5.3.5 How do I get the original `message`?

Ask the command to pass you the invocation context via `pass_context`. This context will be passed as the first parameter.

Example:

```
@bot.command(pass_context=True)
async def joined_at(ctx, member: discord.Member = None):
    if member is None:
        member = ctx.message.author
```

```
    await bot.say('{0} joined at {0.joined_at}'.format(member))
```

5.3.6 How do I make a subcommand?

Use the `group` decorator. This will transform the callback into a `Group` which will allow you to add commands into the group operating as "subcommands". These groups can be arbitrarily nested as well.

Example:

```
@bot.group(pass_context=True)
async def git(ctx):
    if ctx.invoked_subcommand is None:
        await bot.say('Invalid git command passed...')

@git.command()
async def push(remote: str, branch: str):
    await bot.say('Pushing to {} {}'.format(remote, branch))
```

This could then be used as `?git push origin master`.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X