discord.py

v: stable

Interactions API Reference

The following section outlines the API of interactions, as implemented by the library.

For documentation about the rest of the library, check API Reference.

Models

Similar to Discord Models, these are not meant to be constructed by the user.

Interaction

class discord.**Interaction**

Attributes

app_permissions
application_id
channel
channel_id
client
command
command_failed
created_at
data
expires_at
extras
followup
guild
guild_id
guild_locale
id
locale
message
namespace
permissions
response
token
type
user

Methods

async delete_original_response
async edit_original_response
def is_expired
async original_response
async translate

Represents a Discord interaction.

v: stable

An interaction happens when a user does an action that needs to be notified. Current examples are slash commands and components.

New in version 2.0.

id

The interaction's ID.

Type:
`int`

type

The interaction type.

Type:
`InteractionType`

guild_id

The guild ID the interaction was sent from.

Type:
Optional[`int`]

channel

The channel the interaction was sent from.

Note that due to a Discord limitation, if sent from a DM channel `recipient` is `None`.

Type:
Optional[Union[`abc.GuildChannel`, `abc.PrivateChannel`, `Thread`]]

application_id

The application ID that the interaction was for.

Type:
`int`

user

The user or member that sent the interaction.

Type:
Union[`User`, `Member`]

message

The message that sent this interaction.

This is only available for `InteractionType.component` interactions.

Type:
Optional[`Message`]

token

token

The token to continue the interaction. These are valid for 15 minutes.

Type:
str

data

The raw interaction data.

Type:
dict

locale

The locale of the user invoking the interaction.

Type:
Locale

guild_locale

The preferred locale of the guild the interaction was sent from, if any.

Type:
Optional[Locale]

extras

A dictionary that can be used to store extraneous data for use during interaction processing. The library will not touch any values or keys within this dictionary.

Type:
dict

command_failed

Whether the command associated with this interaction failed to execute. This includes checks and execution.

Type:
bool

property client

The client that is handling this interaction.

Note that AutoShardedClient , Bot , and AutoShardedBot are all subclasses of client.

Type:
Client

property guild

The guild the interaction was sent from.

Type:
Optional[`Guild`]

property **channel_id**

The ID of the channel the interaction was sent from.

Type:
Optional[`int`]

property **permissions**

The resolved permissions of the member in the channel, including overwrites.

In a non-guild context where this doesn't apply, an empty permissions object is returned.

Type:
`Permissions`

property **app_permissions**

The resolved permissions of the application or the bot, including overwrites.

Type:
`Permissions`

namespace

The resolved namespace for this interaction.

If the interaction is not an application command related interaction or the client does not have a tree attached to it then this returns an empty namespace.

Type:
`app_commands.Namespace`

command

The command being called from this interaction.

If the interaction is not an application command related interaction or the command is not found in the client's attached tree then `None` is returned.

Type:
Optional[Union[`app_commands.Command` , `app_commands.ContextMenu`]]

response

Returns an object responsible for handling responding to the interaction.

v: stable

A response can only be done once. If secondary messages need to be ser.., consider using `followup` instead.

Type:

Type:

InteractionResponse

followup

Returns the follow up webhook for follow up interactions.

Type:

Webhook

property created_at

When the interaction was created.

Type:

datetime.datetime

property expires_at

When the interaction expires.

Type:

datetime.datetime

is_expired()

bool : Returns `True` if the interaction is expired.

await original_response()

This function is a *coroutine*.

Fetches the original interaction response message associated with the interaction.

If the interaction response was a newly created message (i.e. through `InteractionResponse.send_message()` or `InteractionResponse.defer()`, where `thinking` is `True`) then this returns the message that was sent using that response. Otherwise, this returns the message that triggered the interaction (i.e. through a component).

Repeated calls to this will return a cached value.

Raises:

- HTTPException – Fetching the original response message failed.
- ClientException – The channel for the message could not be resolved.
- NotFound – The interaction response message does not exist.

Returns:

The original interaction response message.

Return type:

InteractionMessage

await **edit_original_response**(***, *content=...*, *embeds=...*, *embed=...*, *attachments=...*, *view=...*, *allowed_mentions=None*)

This function is a *coroutine*.

Edits the original interaction response message.

This is a lower level interface to `InteractionMessage.edit()` in case you do not want to fetch the message and save an HTTP request.

This method is also the only way to edit the original message if the message sent was ephemeral.

Parameters:

- **content** (Optional[`str`]) – The content to edit the message with or `None` to clear it.
- **embeds** (List[`Embed`]) – A list of embeds to edit the message with.
- **embed** (Optional[`Embed`]) – The embed to edit the message with. `None` suppresses the embeds. This should not be mixed with the `embeds` parameter.
- **attachments** (List[Union[`Attachment`, `File`]]) – A list of attachments to keep in the message as well as new files to upload. If `[]` is passed then all attachments are removed.

  > **Note**
  > New files will always appear after current attachments.

- **allowed_mentions** (`AllowedMentions`) – Controls the mentions being processed in this message. See `abc.Messageable.send()` for more information.
- **view** (Optional[`View`]) – The updated view to update this message with. If `None` is passed then the view is removed.

Raises:

- `HTTPException` – Editing the message failed.
- `NotFound` – The interaction response message does not exist.
- `Forbidden` – Edited a message that is not yours.
- `TypeError` – You specified both `embed` and `embeds`
- `ValueError` – The length of `embeds` was invalid.

Returns:

The newly edited message.

Return type:

`InteractionMessage`

await **delete_original_response**()

This function is a *coroutine*

This function is a *coroutine*.

Deletes the original interaction response message.

This is a lower level interface to `InteractionMessage.delete()` in case you do not want to fetch the message and save an HTTP request.

Raises:

- `HTTPException` – Deleting the message failed.
- `NotFound` – The interaction response message does not exist or has already been deleted.
- `Forbidden` – Deleted a message that is not yours.

await **translate**(*string*, ***, *locale=...*, *data=...*)

This function is a *coroutine*.

Translates a string using the set `Translator`.

New in version 2.1.

Parameters:

- **string** (Union[`str`, `locale_str`]) – The string to translate. `locale_str` can be used to add more context, information, or any metadata necessary.
- **locale** (`Locale`) – The locale to use, this is handy if you want the translation for a specific locale. Defaults to the user's `locale`.
- **data** (*Any*) – The extraneous data that is being translated. If not specified, either `command` or `message` will be passed, depending on which is available in the context.

Returns:

The translated string, or `None` if a translator was not set.

Return type:

Optional[`str`]

InteractionResponse

class discord.**InteractionResponse**

Attributes

type

Methods

async autocomplete
async defer
async edit_message
def is_done
async pong

v: stable

async send_message
async send_modal

Represents a Discord interaction response.

This type can be accessed through `Interaction.response`.

New in version 2.0.

`is_done`()

`bool`: Indicates whether an interaction response has been done before.

An interaction can only be responded to once.

property `type`

The type of response that was sent, `None` if response is not done.

Type:
`InteractionResponseType`

await `defer`(*, *ephemeral* = *False*, *thinking* = *False*)

This function is a *coroutine*.

Defers the interaction response.

This is typically used when the interaction is acknowledged and a secondary action will be done later.

This is only supported with the following interaction types:

- `InteractionType.application_command`
- `InteractionType.component`
- `InteractionType.modal_submit`

Parameters:

- **ephemeral** (`bool`) – Indicates whether the deferred message will eventually be ephemeral. This only applies to `InteractionType.application_command` interactions, or if `thinking` is `True`.
- **thinking** (`bool`) – Indicates whether the deferred type should be `InteractionResponseType.deferred_channel_message` instead of the default `InteractionResponseType.deferred_message_update` if both are valid. In UI terms, this is represented as if the bot is thinking of a response. It is your responsibility to eventually send a followup message via `Interaction.followup` to make this thinking state go away. Application commands (AKA Slash commands) cannot use `InteractionResponseType.deferred_message_update`.

Raises:

- HTTPException – Deferring the interaction failed.
- InteractionResponded – This interaction has already been responded to before.

await **pong**( )

This function is a *coroutine*.

Pongs the ping interaction.

This should rarely be used.

Raises:

- HTTPException – Ponging the interaction failed.
- InteractionResponded – This interaction has already been responded to before.

await **send_message**(*content=None*, ***, *embed=...*, *embeds=...*, *file=...*, *files=...*, *view=...*, *tts=False*, *ephemeral=False*, *allowed_mentions=...*, *suppress_embeds=False*, *silent=False*, *delete_after=None*)

This function is a *coroutine*.

Responds to this interaction by sending a message.

Parameters:

- **content** (Optional[str]) – The content of the message to send.
- **embeds** (List[Embed]) – A list of embeds to send with the content. Maximum of 10. This cannot be mixed with the embed parameter.
- **embed** (Embed) – The rich embed for the content to send. This cannot be mixed with embeds parameter.
- **file** (File) – The file to upload.
- **files** (List[File]) – A list of files to upload. Must be a maximum of 10.
- **tts** (bool) – Indicates if the message should be sent using text-to-speech.
- **view** (discord.ui.View) – The view to send with the message.
- **ephemeral** (bool) – Indicates if the message should only be visible to the user who started the interaction. If a view is sent with an ephemeral message and it has no timeout set then the timeout is set to 15 minutes.
- **allowed_mentions** (AllowedMentions) – Controls the mentions being processed in this message. See abc.Messageable.send() for more information.
- **suppress_embeds** (bool) – Whether to suppress embeds for the message. This sends the message without any embeds if set to True .
- **silent** (bool) – Whether to suppress push and desktop notifications for the message. This will increment the mention counter in the UI, but will not actually send a notification.

New in version 2.2

New in version 2.2.

- **delete_after** (`float`) –
  If provided, the number of seconds to wait in the background before deleting the message we just sent. If the deletion fails, then it is silently ignored.
  New in version 2.1.

Raises:

- `HTTPException` – Sending the message failed.
- `TypeError` – You specified both `embed` and `embeds` or `file` and `files`.
- `ValueError` – The length of `embeds` was invalid.
- `InteractionResponded` – This interaction has already been responded to before.

await **edit_message** (**, content=..., embed=..., embeds=..., attachments=..., view=..., allowed_mentions=..., delete_after=None*)

This function is a *coroutine*.

Responds to this interaction by editing the original message of a component or modal interaction.

Parameters:

- **content** (Optional[`str`]) – The new content to replace the message with. `None` removes the content.
- **embeds** (List[`Embed`]) – A list of embeds to edit the message with.
- **embed** (Optional[`Embed`]) – The embed to edit the message with. `None` suppresses the embeds. This should not be mixed with the `embeds` parameter.
- **attachments** (List[Union[`Attachment` , `File`]]) –
  A list of attachments to keep in the message as well as new files to upload. If `[]` is passed then all attachments are removed.

  > **Note**
  > New files will always appear after current attachments.

- **view** (Optional[`View`]) – The updated view to update this message with. If `None` is passed then the view is removed.
- **allowed_mentions** (Optional[`AllowedMentions`]) – Controls the mentions being processed in this message. See `Message.edit()` for more information.
- **delete_after** (`float`) –
  If provided, the number of seconds to wait in the background before deleting the message we just edited. If the deletion fails, then it is silently ignored.
  New in version 2.2.

Raises:

- `HTTPException` – Editing the message failed.

- `TypeError` – You specified both `embed` and `embeds`.
- `InteractionResponded` – This interaction has already been responded to before.

await **send_modal**(*modal*, */*)

This function is a *coroutine*.

Responds to this interaction by sending a modal.

Parameters:

modal (`Modal`) – The modal to send.

Raises:

- `HTTPException` – Sending the modal failed.
- `InteractionResponded` – This interaction has already been responded to before.

await **autocomplete**(*choices*)

This function is a *coroutine*.

Responds to this interaction by giving the user the choices they can use.

Parameters:

choices (List[`Choice`]) – The list of new choices as the user is typing.

Raises:

- `HTTPException` – Sending the choices failed.
- `ValueError` – This interaction cannot respond with autocomplete.
- `InteractionResponded` – This interaction has already been responded to before.

InteractionMessage

class discord.**InteractionMessage**

Attributes

clean_content
created_at
edited_at
jump_url
raw_channel_mentions
raw_mentions
raw_role_mentions
system_content

Methods

async add_files
async add_reaction
async clear_reaction
async clear_reactions
async create_thread
async delete
async edit
async fetch

v: stable

async fetch
def is_system
async pin
async publish
async remove_attachments
async remove_reaction
async reply
def to_reference
async unpin

Represents the original interaction response message.

This allows you to edit or delete the message associated with the interaction response. To retrieve this object see `Interaction.original_response()`.

This inherits from `discord.Message` with changes to `edit()` and `delete()` to work.

New in version 2.0.

await **edit**(**, content=..., embeds=..., embed=..., attachments=..., view=..., allowed_mentions=None, delete_after=None*)

This function is a *coroutine*.

Edits the message.

Parameters:

- **content** (Optional[`str`]) – The content to edit the message with or `None` to clear it.
- **embeds** (List[`Embed`]) – A list of embeds to edit the message with.
- **embed** (Optional[`Embed`]) – The embed to edit the message with. `None` suppresses the embeds. This should not be mixed with the `embeds` parameter.
- **attachments** (List[Union[`Attachment` , `File`]]) –
  A list of attachments to keep in the message as well as new files to upload. If `[]` is passed then all attachments are removed.

  > **Note**
  > New files will always appear after current attachments.

- **allowed_mentions** (`AllowedMentions`) – Controls the mentions being processed in this message. See `abc.Messageable.send()` for more information.
- **view** (Optional[`View`]) – The updated view to update this message with. If `None` is passed then the view is removed.
- **delete_after** (Optional[`float`]) –
  If provided, the number of seconds to wait in the background before deleting the message we just sent. If the deletion fails, then it is silently ignored.
  New in version 2.2.

Raises:

- HTTPException – Editing the message failed.
- Forbidden – Edited a message that is not yours.
- TypeError – You specified both `embed` and `embeds`
- ValueError – The length of `embeds` was invalid.

Returns:

The newly edited message.

Return type:

InteractionMessage

await **add_files**(** files*)

This function is a *coroutine*.

Adds new files to the end of the message attachments.

New in version 2.0.

Parameters:

***files** (File) – New files to add to the message.

Raises:

- HTTPException – Editing the message failed.
- Forbidden – Tried to edit a message that isn't yours.

Returns:

The newly edited message.

Return type:

InteractionMessage

await **remove_attachments**(** attachments*)

This function is a *coroutine*.

Removes attachments from the message.

New in version 2.0.

Parameters:

***attachments** (Attachment) – Attachments to remove from the message.

Raises:

- HTTPException – Editing the message failed.
- Forbidden – Tried to edit a message that isn't yours.

Returns:

The newly edited message.

Return type:

InteractionMessage

await **delete**(**, delay=None*)

This function is a *coroutine*.

Deletes the message.

Parameters:

delay (Optional[float]) – If provided, the number of seconds to wait before deleting the message. The waiting is done in the background and deletion failures are ignored.

Raises:

- Forbidden – You do not have proper permissions to delete the message.
- NotFound – The message was deleted already.
- HTTPException – Deleting the message failed.

await **add_reaction**(*emoji, /*)

This function is a *coroutine*.

Adds a reaction to the message.

The emoji may be a unicode emoji or a custom guild Emoji .

You must have read_message_history to do this. If nobody else has reacted to the message using this emoji, add_reactions is required.

Changed in version 2.0: emoji parameter is now positional-only.

Changed in version 2.0: This function will now raise TypeError instead of InvalidArgument .

Parameters:

emoji (Union[Emoji , Reaction , PartialEmoji , 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 emoji you specified was not found.
- TypeError – The emoji parameter is invalid.

clean_content

A property that returns the content in a “cleaned up” manner. This basicall mentions are transformed into the way the client shows it. e.g. <#id> wil into #name .

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

> **Note**
>
> This *does not* affect markdown. If you want to escape or remove markdown then use `utils.escape_markdown()` or `utils.remove_markdown()` respectively, along with this function.

Type:
`str`

await **clear_reaction**(*emoji*)

This function is a *coroutine*.

Clears a specific reaction from the message.

The emoji may be a unicode emoji or a custom guild `Emoji`.

You must have `manage_messages` to do this.

New in version 1.3.

Changed in version 2.0: This function will now raise `TypeError` instead of `InvalidArgument`.

Parameters:
emoji (Union[`Emoji`, `Reaction`, `PartialEmoji`, `str`]) – The emoji to clear.

Raises:
- `HTTPException` – Clearing the reaction failed.
- `Forbidden` – You do not have the proper permissions to clear the reaction.
- `NotFound` – The emoji you specified was not found.
- `TypeError` – The emoji parameter is invalid.

await **clear_reactions**()

This function is a *coroutine*.

Removes all the reactions from the message.

You must have `manage_messages` to do this.

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

await **create_thread**(*, *name*, *auto_archive_duration*=..., *slowmode_delay*=*None*, *reason*=*None*)

slowmode_delay = None, reason = None)

This function is a *coroutine*.

Creates a public thread from this message.

You must have `create_public_threads` in order to create a public thread from a message.

The channel this message belongs in must be a `TextChannel`.

New in version 2.0.

Parameters:

- **name** (`str`) – The name of the thread.
- **auto_archive_duration** (`int`) –
  The duration in minutes before a thread is automatically hidden from the channel list. If not provided, the channel's default auto archive duration is used.
  Must be one of `60`, `1440`, `4320`, or `10080`, if provided.
- **slowmode_delay** (Optional[`int`]) – Specifies the slowmode rate limit for user in this channel, in seconds. The maximum value possible is `21600`. By default no slowmode rate limit if this is `None`.
- **reason** (Optional[`str`]) – The reason for creating a new thread. Shows up on the audit log.

Raises:

- `Forbidden` – You do not have permissions to create a thread.
- `HTTPException` – Creating the thread failed.
- `ValueError` – This message does not have guild info attached.

Returns:

The created thread.

Return type:

`Thread`

property **created_at**

The message's creation time in UTC.

Type:

`datetime.datetime`

property **edited_at**

An aware UTC datetime object containing the edited time of the message.

Type:

Optional[`datetime.datetime`]

v: stable

await **fetch**()

This function is a *coroutine*.

Fetches the partial message to a full `Message`.

Raises:

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

Returns:

The full message.

Return type:

`Message`

`is_system()`

`bool` : Whether the message is a system message.

A system message is a message that is constructed entirely by the Discord API in response to something.

New in version 1.3.

property `jump_url`

Returns a URL that allows the client to jump to this message.

Type:

`str`

await `pin(*, reason=None)`

This function is a *coroutine*.

Pins the message.

You must have `manage_messages` to do this in a non-private channel context.

Parameters:

reason (Optional[`str`]) –
The reason for pinning the message. Shows up on the audit log.
New in version 1.4.

Raises:

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

await `publish()`

This function is a *coroutine*.

Publishes this message to the channel's followers.

The message must have been sent in a news channel. You must have `send_messages` to do this.

If the message is not your own then `manage_messages` is also needed.

Raises:

- `Forbidden` – You do not have the proper permissions to publish this message or the channel is not a news channel.
- `HTTPException` – Publishing the message failed.

`raw_channel_mentions`

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

Type:

List[`int`]

`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 to receive the user IDs of mentioned users even in a private message context.

Type:

List[`int`]

`raw_role_mentions`

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

Type:

List[`int`]

await **remove_reaction**(*emoji*, *member*)

This function is a *coroutine*.

Remove a reaction by the member from the message.

The emoji may be a unicode emoji or a custom guild `Emoji`.

If the reaction is not your own (i.e. `member` parameter is not you) then `manage_messages` is needed.

v: stable

The `member` parameter must represent a member and meet the `abc.Snowflake` abc

The "member" parameter must represent a member and meet the abc.Snowflake abc.

Changed in version 2.0: This function will now raise `TypeError` instead of `InvalidArgument`.

Parameters:

- **emoji** (Union[`Emoji`, `Reaction`, `PartialEmoji`, `str`]) – The emoji to remove.
- **member** (`abc.Snowflake`) – The member for which to remove the reaction.

Raises:

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

await **reply**(*content=None*, ***kwargs*)

This function is a *coroutine*.

A shortcut method to `abc.Messageable.send()` to reply to the `Message`.

New in version 1.6.

Changed in version 2.0: This function will now raise `TypeError` or `ValueError` instead of `InvalidArgument`.

Raises:

- `HTTPException` – Sending the message failed.
- `Forbidden` – You do not have the proper permissions to send the message.
- `ValueError` – The `files` list is not of the appropriate size
- `TypeError` – You specified both `file` and `files`.

Returns:

The message that was sent.

Return type:

`Message`

system_content

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

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

Type:

`str`

v: stable

to_reference(*, *fail_if_not_exists=True*)

Creates a `MessageReference` from the current message.

New in version 1.6.

Parameters:

fail_if_not_exists (`bool`) –
Whether replying using the message reference should raise `HTTPException` if the message no longer exists or Discord could not fetch the message.
New in version 1.7.

Returns:

The reference to this message.

Return type:

`MessageReference`

await **unpin** (*, *reason=None*)

This function is a *coroutine*.

Unpins the message.

You must have `manage_messages` to do this in a non-private channel context.

Parameters:

reason (Optional[`str`]) –
The reason for unpinning the message. Shows up on the audit log.
New in version 1.4.

Raises:

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

MessageInteraction

class discord. **MessageInteraction**

Attributes

created_at
id
name
type
user

v: stable

Represents the interaction that a `Message` is a response to.

New in version 2.0.

Supported Operations

`x == y`

Checks if two message interactions are equal.

`x != y`

Checks if two message interactions are not equal.

`hash(x)`

Returns the message interaction's hash.

`id`

The interaction ID.

Type:
`int`

`type`

The interaction type.

Type:
`InteractionType`

`name`

The name of the interaction.

Type:
`str`

`user`

The user or member that invoked the interaction.

Type:
Union[`User`, `Member`]

property `created_at`

The interaction's creation time in UTC.

Type:
`datetime.datetime`

Component

class discord.**Component**

Attributes

type

Represents a Discord Bot UI Kit Component.

Currently, the only components supported by Discord are:

- ActionRow
- Button
- SelectMenu
- TextInput

This class is abstract and cannot be instantiated.

New in version 2.0.

property **type**

The type of component.

Type:
ComponentType

ActionRow

class discord.**ActionRow**

Attributes

children
type

Represents a Discord Bot UI Kit Action Row.

This is a component that holds up to 5 children components in a row.

This inherits from Component .

New in version 2.0.

children

The children components that this holds, if any.

Type:
List[Union[Button , SelectMenu , TextInput]]

property **type**

The type of component.

Type:
ComponentType

Button

class discord.**Button**

Attributes

custom_id
disabled
emoji
label
style
type
url

Represents a button from the Discord Bot UI Kit.

This inherits from Component .

Note

The user constructible and usable type to create a button is discord.ui.Button not this one.

New in version 2.0.

style

The style of the button.

Type:
ButtonStyle

custom_id

The ID of the button that gets received during an interaction. If this button is for a URL, it does not have a custom ID.

Type:
Optional[str]

url

The URL this button sends you to.

Type:

Optional[str]

disabled

Whether the button is disabled or not.

Type:

bool

label

The label of the button, if any.

Type:

Optional[str]

emoji

The emoji of the button, if available.

Type:

Optional[PartialEmoji]

property **type**

The type of component.

Type:

ComponentType

SelectMenu

class discord.**SelectMenu**

Attributes

channel_types
custom_id
disabled
max_values
min_values
options
placeholder
type

Represents a select menu from the Discord Bot UI Kit.

A select menu is functionally the same as a dropdown, however on mobile it r v: stable differently.

Note

The user constructible and usable type to create a select menu is `discord.ui.Select` not this one.

New in version 2.0.

type

The type of component.

Type:
`ComponentType`

custom_id

The ID of the select menu that gets received during an interaction.

Type:
Optional[`str`]

placeholder

The placeholder text that is shown if nothing is selected, if any.

Type:
Optional[`str`]

min_values

The minimum number of items that must be chosen for this select menu. Defaults to 1 and must be between 0 and 25.

Type:
`int`

max_values

The maximum number of items that must be chosen for this select menu. Defaults to 1 and must be between 1 and 25.

Type:
`int`

options

A list of options that can be selected in this menu.

Type:
List[`SelectOption`]

disabled

Whether the select is disabled or not.

Type:

bool

channel_types

A list of channel types that are allowed to be chosen in this select menu.

Type:

List[ChannelType]

TextInput

class discord.**TextInput**

Attributes

custom_id
default
label
max_length
min_length
placeholder
required
style
type
value

Represents a text input from the Discord Bot UI Kit.

> **Note**
>
> The user constructible and usable type to create a text input is discord.ui.TextInput not this one.

New in version 2.0.

custom_id

The ID of the text input that gets received during an interaction.

Type:

Optional[str]

label

The label to display above the text input.

Type:

str

v: stable

style

style

The style of the text input.

Type:
TextStyle

placeholder

The placeholder text to display when the text input is empty.

Type:
Optional[str]

value

The default value of the text input.

Type:
Optional[str]

required

Whether the text input is required.

Type:
bool

min_length

The minimum length of the text input.

Type:
Optional[int]

max_length

The maximum length of the text input.

Type:
Optional[int]

property type

The type of component.

Type:
ComponentType

property default

The default value of the text input.

This is an alias to value .

Type:
Optional[str]

AppCommand

class discord.app_commands.**AppCommand**

Attributes

application_id
default_member_permissions
description
description_localizations
dm_permission
guild
guild_id
id
mention
name
name_localizations
nsfw
options
type

Methods

async delete
async edit
async fetch_permissions

Represents an application command.

In common parlance this is referred to as a “Slash Command” or a “Context Menu Command”.

New in version 2.0.

Supported Operations

`x == y`

Checks if two application commands are equal.

`x != y`

Checks if two application commands are not equal.

`hash(x)`

Returns the application command’s hash.

`str(x)`

Returns the application command’s name.

v: stable

`id`

The application command's ID.

Type:
`int`

application_id

The application command's application's ID.

Type:
`int`

type

The application command's type.

Type:
`AppCommandType`

name

The application command's name.

Type:
`str`

description

The application command's description.

Type:
`str`

name_localizations

The localised names of the application command. Used for display purposes.

Type:
Dict[`Locale` , `str`]

description_localizations

The localised descriptions of the application command. Used for display purposes.

Type:
Dict[`Locale` , `str`]

options

A list of options.

Type:
List[Union[`Argument` , `AppCommandGroup`]]

default_member_permissions

The default member permissions that can run this command.

Type:
Optional[Permissions]

dm_permission

A boolean that indicates whether this command can be run in direct messages.

Type:
bool

guild_id

The ID of the guild this command is registered in. A value of None denotes that it is a global command.

Type:
Optional[int]

nsfw

Whether the command is NSFW and should only work in NSFW channels.

Type:
bool

property mention

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

Type:
str

property guild

Returns the guild this command is registered to if it exists.

Type:
Optional[Guild]

await delete( )

This function is a *coroutine*.

Deletes the application command.

Raises:

- NotFound – The application command was not found.
- Forbidden – You do not have permission to delete this application command.
- HTTPException – Deleting the application command failed.
- MissingApplicationID – The client does not have an application ID.

await edit(*, name=, description=

default_member_permissions = ..., *dm_permission* = ..., *options* = ...)

This function is a *coroutine*.

Edits the application command.

Parameters:

- **name** (`str`) – The new name for the application command.
- **description** (`str`) – The new description for the application command.
- **default_member_permissions** (Optional[`Permissions`]) – The new default permissions needed to use this application command. Pass value of `None` to remove any permission requirements.
- **dm_permission** (`bool`) – Indicates if the application command can be used in DMs.
- **options** (List[Union[`Argument` , `AppCommandGroup`]]) – List of new options for this application command.

Raises:

- `NotFound` – The application command was not found.
- `Forbidden` – You do not have permission to edit this application command.
- `HTTPException` – Editing the application command failed.
- `MissingApplicationID` – The client does not have an application ID.

Returns:

The newly edited application command.

Return type:

`AppCommand`

await **fetch_permissions** (*guild*)

This function is a *coroutine*.

Retrieves this command's permission in the guild.

Parameters:

guild (`Snowflake`) – The guild to retrieve the permissions from.

Raises:

- `Forbidden` – You do not have permission to fetch the application command's permissions.
- `HTTPException` – Fetching the application command's permissions failed.
- `MissingApplicationID` – The client does not have an application ID.
- `NotFound` – The application command's permissions could not be v: stable can also indicate that the permissions are synced with the guild (i.e. they are unchanged from the default).

Returns:

An object representing the application command's permissions in the guild.

Return type:

GuildAppCommandPermissions

AppCommandGroup

class discord.app_commands.**AppCommandGroup**

Attributes

description
description_localizations
mention
name
name_localizations
options
parent
qualified_name
type

Represents an application command subcommand.

New in version 2.0.

type

The type of subcommand.

Type:

AppCommandOptionType

name

The name of the subcommand.

Type:

str

description

The description of the subcommand.

Type:

str

name_localizations

The localised names of the subcommand. Used for display purposes.

Type:

Type:
Dict[Locale , str]

description_localizations

The localised descriptions of the subcommand. Used for display purposes.

Type:
Dict[Locale , str]

options

A list of options.

Type:
List[Union[Argument , AppCommandGroup]]

parent

The parent application command.

Type:
Union[AppCommand , AppCommandGroup]

property qualified_name

Returns the fully qualified command name.

The qualified name includes the parent name as well. For example, in a command like `/foo bar` the qualified name is `foo bar`.

Type:
str

property mention

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

Type:
str

AppCommandChannel

class discord.app_commands.**AppCommandChannel**

Attributes

created_at
guild
guild_id
id
mention

Methods

async fetch
def resolve

v: stable

> name
> permissions
> type

Represents an application command partially resolved channel object.

New in version 2.0.

Supported Operations

`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.

`id`

The ID of the channel.

Type:
`int`

`type`

The type of channel.

Type:
`ChannelType`

`name`

The name of the channel.

Type:
`str`

`permissions`

The resolved permissions of the user who invoked the application command in that channel.

Type:
`Permissions`

guild_id

The guild ID this channel belongs to.

Type:
int

property **guild**

The channel's guild, from cache, if found.

Type:
Optional[Guild]

resolve( )

Resolves the application command channel to the appropriate channel from cache if found.

Returns:
The resolved guild channel or None if not found in cache.

Return type:
Optional[abc.GuildChannel]

await **fetch( )**

This function is a *coroutine*.

Fetches the partial channel to a full abc.GuildChannel .

Raises:
- NotFound – The channel was not found.
- Forbidden – You do not have the permissions required to get a channel.
- HTTPException – Retrieving the channel failed.

Returns:
The full channel.

Return type:
abc.GuildChannel

property **mention**

The string that allows you to mention the channel.

Type:
str

property **created_at**

v: stable

An aware timestamp of when this channel was created in UTC.

Type:

type:
datetime.datetime

AppCommandThread

class discord.app_commands.**AppCommandThread**

Attributes

archive_timestamp
archived
archiver_id
auto_archive_duration
created_at
guild
guild_id
id
invitable
locked
mention
name
parent
parent_id
permissions
type

Methods

async fetch
def resolve

Represents an application command partially resolved thread object.

New in version 2.0.

Supported Operations

x == y

Checks if two thread are equal.

x != y

Checks if two thread are not equal.

hash(x)

Returns the thread's hash.

str(x)

Returns the thread's name.

v: stable

id

__

The ID of the thread.

Type:

`int`

type

The type of thread.

Type:

`ChannelType`

name

The name of the thread.

Type:

`str`

parent_id

The parent text channel ID this thread belongs to.

Type:

`int`

permissions

The resolved permissions of the user who invoked the application command in that thread.

Type:

`Permissions`

guild_id

The guild ID this thread belongs to.

Type:

`int`

archived

Whether the thread is archived.

Type:

`bool`

locked

Whether the thread is locked.

Type:

`bool`

invitable

Whether non-moderators can add other non-moderators to this thread. This is always `True` for public threads.

Type:
`bool`

archiver_id

The user's ID that archived this thread.

Type:
Optional[`int`]

auto_archive_duration

The duration in minutes until the thread is automatically hidden from the channel list. Usually a value of 60, 1440, 4320 and 10080.

Type:
`int`

archive_timestamp

An aware timestamp of when the thread's archived status was last updated in UTC.

Type:
`datetime.datetime`

property guild

The channel's guild, from cache, if found.

Type:
Optional[`Guild`]

property parent

The parent channel this thread belongs to.

Type:
Optional[`TextChannel`]

property mention

The string that allows you to mention the thread.

Type:
`str`

property created_at

An aware timestamp of when the thread was created in UTC.

Note

> **Note**
>
> This timestamp only exists for threads created after 9 January 2022, otherwise returns `None`.

`resolve`()

Resolves the application command channel to the appropriate channel from cache if found.

Returns:

The resolved guild channel or `None` if not found in cache.

Return type:

Optional[`abc.GuildChannel`]

`await` `fetch`()

This function is a *coroutine*.

Fetches the partial channel to a full `Thread`.

Raises:

- `NotFound` – The thread was not found.
- `Forbidden` – You do not have the permissions required to get a thread.
- `HTTPException` – Retrieving the thread failed.

Returns:

The full thread.

Return type:

`Thread`

AppCommandPermissions

class discord.app_commands.**AppCommandPermissions**

Attributes

guild
id
permission
target
type

Represents the permissions for an application command.

v: stable

New in version 2.0.

`guild`

guild

The guild associated with this permission.

Type:
`Guild`

id

The ID of the permission target, such as a role, channel, or guild. The special `guild_id - 1` sentinel is used to represent “all channels”.

Type:
`int`

target

The role, user, or channel associated with this permission. This could also be the `AllChannels` sentinel type. Falls back to `Object` if the target could not be found in the cache.

Type:
Any

type

The type of permission.

Type:
`AppCommandPermissionType`

permission

The permission value. `True` for allow, `False` for deny.

Type:
`bool`

GuildAppCommandPermissions

class discord.app_commands. **GuildAppCommandPermissions**

Attributes

application_id
command
guild
guild_id
id
permissions

v: stable

Represents the permissions for an application command in a guild.

New in version 2.0.

application_id

The application ID.

Type:
int

command

The application command associated with the permissions.

Type:
AppCommand

id

ID of the command or the application ID. When this is the application ID instead of a command ID, the permissions apply to all commands that do not contain explicit overwrites.

Type:
int

guild_id

The guild ID associated with the permissions.

Type:
int

permissions

The permissions, this is a max of 100.

Type:
List[AppCommandPermissions]

property **guild**

The guild associated with the permissions.

Type:
Guild

Argument

class discord.app_commands.**Argument**

v: stable

Attributes

> autocomplete
> channel_types
> choices
> description
> description_localizations
> max_length
> max_value
> min_length
> min_value
> name
> name_localizations
> parent
> required
> type

Represents an application command argument.

New in version 2.0.

type

The type of argument.

Type:
`AppCommandOptionType`

name

The name of the argument.

Type:
`str`

description

The description of the argument.

Type:
`str`

name_localizations

The localised names of the argument. Used for display purposes.

Type:
Dict[`Locale`, `str`]

description_localizations

The localised descriptions of the argument. Used for display purposes.

Type:
Dict[`Locale`, `str`]

required

Whether the argument is required.

Type:
bool

choices

A list of choices for the command to choose from for this argument.

Type:
List[Choice]

parent

The parent application command that has this argument.

Type:
Union[AppCommand , AppCommandGroup]

channel_types

The channel types that are allowed for this parameter.

Type:
List[ChannelType]

min_value

The minimum supported value for this parameter.

Type:
Optional[Union[int , float]]

max_value

The maximum supported value for this parameter.

Type:
Optional[Union[int , float]]

min_length

The minimum allowed length for this parameter.

Type:
Optional[int]

max_length

The maximum allowed length for this parameter.

Type:
Optional[int]

autocomplete

Whether the argument has autocomplete.

Type:
bool

AllChannels

class discord.app_commands.**AllChannels**

Attributes
guild
id

Represents all channels for application command permissions.

New in version 2.0.

guild

The guild the application command permission is for.

Type:
Guild

property **id**

The ID sentinel used to represent all channels. Equivalent to the guild's ID minus 1.

Type:
int

Data Classes

Similar to Data Classes, these can be received and constructed by users.

SelectOption

class discord.**SelectOption**(***, *label*, *value=...*, *description=None*, *emoji=None*, *default=False*)

Attributes
default
description
emoji

label
value

Represents a select menu's option.

These can be created by users.

New in version 2.0.

Parameters:

- **label** (`str`) – The label of the option. This is displayed to users. Can only be up to 100 characters.
- **value** (`str`) – The value of the option. This is not displayed to users. If not provided when constructed then it defaults to the label. Can only be up to 100 characters.
- **description** (Optional[`str`]) – An additional description of the option, if any. Can only be up to 100 characters.
- **emoji** (Optional[Union[`str` , `Emoji` , `PartialEmoji`]]) – The emoji of the option, if available.
- **default** (`bool`) – Whether this option is selected by default.

`label`

The label of the option. This is displayed to users. Can only be up to 100 characters.

Type:
`str`

`value`

The value of the option. This is not displayed to users. If not provided when constructed then it defaults to the label. Can only be up to 100 characters.

Type:
`str`

`description`

An additional description of the option, if any. Can only be up to 100 characters.

Type:
Optional[`str`]

`default`

Whether this option is selected by default.

Type:
`bool`

property `emoji`

The emoji of the option, if available.

Type:
Optional[PartialEmoji]

Choice

class discord.app_commands. **Choice** (**, name, value*)

Represents an application command argument choice.

New in version 2.0.

Supported Operations

x == y

Checks if two choices are equal.

x != y

Checks if two choices are not equal.

hash(x)

Returns the choice's hash.

Parameters:

- **name** (Union[str , locale_str]) – The name of the choice. Used for display purposes. Can only be up to 100 characters.
- **name_localizations** (Dict[Locale , str]) – The localised names of the choice. Used for display purposes.
- **value** (Union[int , str , float]) – The value of the choice. If it's a string, it can only be up to 100 characters long.

Enumerations

class discord. **InteractionType**

Specifies the type of Interaction .

New in version 2.0.

v: stable

ping

Represents Discord pinging to see if the interaction response server is alive.

application_command

Represents a slash command interaction.

component

Represents a component based interaction, i.e. using the Discord Bot UI Kit.

autocomplete

Represents an auto complete interaction.

modal_submit

Represents submission of a modal interaction.

class discord.**InteractionResponseType**

Specifies the response type for the interaction.

New in version 2.0.

pong

Pongs the interaction when given a ping.

See also `InteractionResponse.pong()`

channel_message

Respond to the interaction with a message.

See also `InteractionResponse.send_message()`

deferred_channel_message

Responds to the interaction with a message at a later time.

See also `InteractionResponse.defer()`

deferred_message_update

Acknowledges the component interaction with a promise that the message will update later (though there is no need to actually update the message).

See also `InteractionResponse.defer()`

message_update

Responds to the interaction by editing the message.

See also `InteractionResponse.edit_message()`

autocomplete_result

Responds to the autocomplete interaction with suggested choices.

See also `InteractionResponse.autocomplete()`

modal

Responds to the interaction with a modal.

See also `InteractionResponse.send_modal()`

class discord.**ComponentType**

Represents the component type of a component.

New in version 2.0.

action_row

Represents the group component which holds different components in a row.

button

Represents a button component.

text_input

Represents a text box component.

select

Represents a select component.

string_select

An alias to `select`. Represents a default select component.

user_select

Represents a user select component.

role_select

Represents a role select component.

mentionable_select

Represents a select in which both users and roles can be selected.

class discord.**ButtonStyle**

v: stable

Represents the style of the button component.

New in version 2.0.

primary

Represents a blurple button for the primary action.

secondary

Represents a grey button for the secondary action.

success

Represents a green button for a successful action.

danger

Represents a red button for a dangerous action.

link

Represents a link button.

blurple

An alias for `primary`.

grey

An alias for `secondary`.

gray

An alias for `secondary`.

green

An alias for `success`.

red

An alias for `danger`.

url

An alias for `link`.

class discord.**TextStyle**

Represents the style of the text box component.

New in version 2.0.

short

Represents a short text box.

paragraph

Represents a long form text box.

long

An alias for `paragraph`.

class discord.**AppCommandOptionType**

The application command's option type. This is usually the type of parameter an application command takes.

New in version 2.0.

subcommand

A subcommand.

subcommand_group

A subcommand group.

string

A string parameter.

integer

A integer parameter.

boolean

A boolean parameter.

user

A user parameter.

channel

A channel parameter.

role

A role parameter.

mentionable

A mentionable parameter.

number

A number parameter.

attachment

An attachment parameter.

class discord.**AppCommandType**

The type of application command.

New in version 2.0.

chat_input

A slash command.

user

A user context menu command.

message

A message context menu command.

class discord.**AppCommandPermissionType**

The application command's permission type.

New in version 2.0.

role

The permission is for a role.

channel

The permission is for one or all channels.

user

The permission is for a user.

Bot UI Kit

The library has helpers to aid in creating component-based UIs. These are all in the `discord.ui` package.

View

v: stable

class discord.ui.**View**(**, timeout=180.0*)

Attributes

children
timeout

Methods

cls View.from_message
def add_item
def clear_items
async interaction_check
def is_dispatching
def is_finished
def is_persistent
async on_error
async on_timeout
def remove_item
def stop
async wait

Represents a UI view.

This object must be inherited to create a UI within Discord.

New in version 2.0.

Parameters:

timeout (Optional[`float`]) – Timeout in seconds from last interaction with the UI before no longer accepting input. If `None` then there is no timeout.

property **timeout**

The timeout in seconds from last interaction with the UI before no longer accepting input. If `None` then there is no timeout.

Type:

Optional[`float`]

property **children**

The list of children attached to this view.

Type:

List[`Item`]

classmethod **from_message**(*message*, */*, ***, *timeout = 180.0*)

Converts a message's components into a `View` .

The `Message.components` of a message are read-only and separate types from those in the `discord.ui` namespace. In order to modify and edit message components they must be converted into a `View` first.

Parameters:

- **message** (`discord.Message`) – The message with components to convert into a view.

- **timeout** (Optional[`float`]) – The timeout of the converted view.

Returns:

The converted view. This always returns a `View` and not one of its subclasses.

Return type:

`View`

`add_item` (*item*)

Adds an item to the view.

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

Parameters:

item (`Item`) – The item to add to the view.

Raises:

- `TypeError` – An `Item` was not passed.
- `ValueError` – Maximum number of children has been exceeded (25) or the row the item is trying to be added to is full.

`remove_item` (*item*)

Removes an item from the view.

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

Parameters:

item (`Item`) – The item to remove from the view.

`clear_items` ( )

Removes all items from the view.

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

await `interaction_check` (*interaction*, */*)

This function is a *coroutine*.

A callback that is called when an interaction happens within the view that checks whether the view should process item callbacks for the interaction.

This is useful to override if, for example, you want to ensure that the interaction author is a given user.

The default implementation of this returns `True` .

> **Note** v: stable
>
> If an exception occurs within the body then the check is considered a failure and

`on_error()` is called.

Parameters:

interaction (`Interaction`) – The interaction that occurred.

Returns:

Whether the view children's callbacks should be called.

Return type:

`bool`

await **on_timeout** ( )

This function is a *coroutine*.

A callback that is called when a view's timeout elapses without being explicitly stopped.

await **on_error** (*interaction*, *error*, *item*, */*)

This function is a *coroutine*.

A callback that is called when an item's callback or `interaction_check()` fails with an error.

The default implementation logs to the library logger.

Parameters:

- **interaction** (`Interaction`) – The interaction that led to the failure.
- **error** (`Exception`) – The exception that was raised.
- **item** (`Item`) – The item that failed the dispatch.

stop ( )

Stops listening to interaction events from this view.

This operation cannot be undone.

is_finished ( )

`bool` : Whether the view has finished interacting.

is_dispatching ( )

`bool` : Whether the view has been added for dispatching purposes.

is_persistent ( )

`bool` : Whether the view is set up as persistent.

A persistent view has all their components with a set `custom_id` and a `t` v: stable `o` `None` .

await **wait** ( )

await **wait**()

This function is a *coroutine*.

Waits until the view has finished interacting.

A view is considered finished when `stop()` is called or it times out.

Returns:

If `True`, then the view timed out. If `False` then the view finished normally.

Return type:

`bool`

Modal

class discord.ui.**Modal**(***, *title=...*, *timeout=None*, *custom_id=...*)

Attributes

children
custom_id
timeout
title

Methods

cls Modal.from_message
def add_item
def clear_items
async interaction_check
def is_dispatching
def is_finished
def is_persistent
async on_error
async on_submit
async on_timeout
def remove_item
def stop
async wait

Represents a UI modal.

This object must be inherited to create a modal popup window within discord.

New in version 2.0.

Examples

```
import discord
from discord import ui

class Questionnaire(ui.Modal, title='Questionnaire Response'):
```

v: stable

```
    name = ui.TextInput(label='Name')
    answer = ui.TextInput(label='Answer', style=discord.TextStyle.paragrap

    async def on_submit(self, interaction: discord.Interaction):
        await interaction.response.send_message(f'Thanks for your response
```

Parameters:

- **title** (`str`) – The title of the modal. Can only be up to 45 characters.
- **timeout** (Optional[`float`]) – Timeout in seconds from last interaction with the UI before no longer accepting input. If `None` then there is no timeout.
- **custom_id** (`str`) – The ID of the modal that gets received during an interaction. If not given then one is generated for you. Can only be up to 100 characters.

`title`

The title of the modal.

Type:
`str`

`custom_id`

The ID of the modal that gets received during an interaction.

Type:
`str`

await **on_submit** (*interaction*, */*)

This function is a *coroutine*.

Called when the modal is submitted.

Parameters:
interaction (`Interaction`) – The interaction that submitted this modal.

await **on_error** (*interaction*, *error*, */*)

This function is a *coroutine*.

A callback that is called when `on_submit()` fails with an error.

The default implementation logs to the library logger.

Parameters:

- **interaction** (`Interaction`) – The interaction that led to the failure.
- **error** (`Exception`) – The exception that was raised.

add_item (*item*)

Adds an item to the view.

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

Parameters:

item (`Item`) – The item to add to the view.

Raises:

- `TypeError` – An `Item` was not passed.
- `ValueError` – Maximum number of children has been exceeded (25) or the row the item is trying to be added to is full.

property **children**

The list of children attached to this view.

Type:

List[`Item`]

clear_items ( )

Removes all items from the view.

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

classmethod **from_message** (*message*, */*, ***, *timeout = 180.0*)

Converts a message's components into a `View` .

The `Message.components` of a message are read-only and separate types from those in the `discord.ui` namespace. In order to modify and edit message components they must be converted into a `View` first.

Parameters:

- **message** (`discord.Message`) – The message with components to convert into a view.
- **timeout** (Optional[`float`]) – The timeout of the converted view.

Returns:

The converted view. This always returns a `View` and not one of its subclasses.

Return type:

`View`

await **interaction_check** (*interaction*, */*)

This function is a *coroutine*.

A callback that is called when an interaction happens within the view that checks whether the view should process item callbacks for the interaction.

This is useful to override if, for example, you want to ensure that the interaction author is a given user.

The default implementation of this returns `True`.

> **Note**
>
> If an exception occurs within the body then the check is considered a failure and `on_error()` is called.

Parameters:
interaction (`Interaction`) – The interaction that occurred.

Returns:
Whether the view children's callbacks should be called.

Return type:
`bool`

`is_dispatching`()

`bool`: Whether the view has been added for dispatching purposes.

`is_finished`()

`bool`: Whether the view has finished interacting.

`is_persistent`()

`bool`: Whether the view is set up as persistent.

A persistent view has all their components with a set `custom_id` and a `timeout` set to `None`.

await `on_timeout`()

This function is a *coroutine*.

A callback that is called when a view's timeout elapses without being explicitly stopped.

`remove_item`(*item*)

Removes an item from the view.

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

Parameters:
item (`Item`) – The item to remove from the view.

`stop`()

Stops listening to interaction events from this view.

This operation cannot be undone.

property `timeout`

property **timeout**

The timeout in seconds from last interaction with the UI before no longer accepting input. If `None` then there is no timeout.

Type:
Optional[`float`]

await **wait** ( )

This function is a *coroutine*.

Waits until the view has finished interacting.

A view is considered finished when `stop()` is called or it times out.

Returns:
If `True` , then the view timed out. If `False` then the view finished normally.

Return type:
`bool`

Item

class discord.ui. **Item**

Attributes
view

Methods
async callback

Represents the base UI item that all UI components inherit from.

The current UI items supported are:

- `discord.ui.Button`
- `discord.ui.Select`
- `discord.ui.TextInput`

New in version 2.0.

property **view**

The underlying view for this item.

Type:
Optional[`View`]

await **callback** (*interaction*)

This function is a *coroutine*.

The callback associated with this UI item.

This can be overridden by subclasses.

Parameters:

interaction (`Interaction`) – The interaction that triggered this UI item.

Button

class discord.ui.**Button**(**, style=<ButtonStyle.secondary: 2>, label=None, disabled=False, custom_id=None, url=None, emoji=None, row=None*)

Attributes

custom_id
disabled
emoji
label
style
url
view

Methods

async callback

Represents a UI button.

New in version 2.0.

Parameters:

- **style** (`discord.ButtonStyle`) – The style of the button.
- **custom_id** (Optional[`str`]) – The ID of the button that gets received during an interaction. If this button is for a URL, it does not have a custom ID.
- **url** (Optional[`str`]) – The URL this button sends you to.
- **disabled** (`bool`) – Whether the button is disabled or not.
- **label** (Optional[`str`]) – The label of the button, if any.
- **emoji** (Optional[Union[`PartialEmoji` , `Emoji` , `str`]]) – The emoji of the button, if available.
- **row** (Optional[`int`]) – The relative row this button belongs to. A Discord component can only have 5 rows. By default, items are arranged automatically into those 5 rows. If you'd like to control the relative positioning of the row then passing an index is advised. For example, row=1 will show up before row=2. Defaults to `None` , which is automatic ordering. The row number must be between 0 and 4 (i.e. zero indexed).

property **style**

The style of the button.

Type:

discord.ButtonStyle

property **custom_id**

The ID of the button that gets received during an interaction.

If this button is for a URL, it does not have a custom ID.

Type:
Optional[str]

property **url**

The URL this button sends you to.

Type:
Optional[str]

property **disabled**

Whether the button is disabled or not.

Type:
bool

property **label**

The label of the button, if available.

Type:
Optional[str]

property **emoji**

The emoji of the button, if available.

Type:
Optional[PartialEmoji]

await **callback** (*interaction*)

This function is a *coroutine*.

The callback associated with this UI item.

This can be overridden by subclasses.

Parameters:
interaction (Interaction) – The interaction that triggered this UI item.

property **view**

The underlying view for this item.

Type:
Optional[View]

`@discord.ui.button(*, label=None, custom_id=None, disabled=False, style=<ButtonStyle.secondary: 2>, emoji=None, row=None)`

A decorator that attaches a button to a component.

The function being decorated should have three parameters, `self` representing the `discord.ui.View`, the `discord.Interaction` you receive and the `discord.ui.Button` being pressed.

> **Note**
>
> Buttons with a URL cannot be created with this function. Consider creating a `Button` manually instead. This is because buttons with a URL do not have a callback associated with them since Discord does not do any processing with it.

Parameters:

- **label** (Optional[`str`]) – The label of the button, if any.
- **custom_id** (Optional[`str`]) – The ID of the button that gets received during an interaction. It is recommended not to set this parameter to prevent conflicts.
- **style** (`ButtonStyle`) – The style of the button. Defaults to `ButtonStyle.grey`.
- **disabled** (`bool`) – Whether the button is disabled or not. Defaults to `False`.
- **emoji** (Optional[Union[`str`, `Emoji`, `PartialEmoji`]]) – The emoji of the button. This can be in string form or a `PartialEmoji` or a full `Emoji`.
- **row** (Optional[`int`]) – The relative row this button belongs to. A Discord component can only have 5 rows. By default, items are arranged automatically into those 5 rows. If you'd like to control the relative positioning of the row then passing an index is advised. For example, row=1 will show up before row=2. Defaults to `None`, which is automatic ordering. The row number must be between 0 and 4 (i.e. zero indexed).

Select Menus

The library provides classes to help create the different types of select menus.

Select

`class discord.ui.Select(*, custom_id=..., placeholder=None, min_values=1, max_values=1, options=..., disabled=False, row=None)`

v: stable

Attributes

custom_id

Methods

def add_option

```
disabled
max_values
min_values
options
placeholder
type
values
view
```

```
def add_option
def append_option
async callback
```

Represents a UI select menu with a list of custom options. This is represented to the user as a dropdown menu.

New in version 2.0.

Parameters:

- **custom_id** (`str`) – The ID of the select menu that gets received during an interaction. If not given then one is generated for you.
- **placeholder** (Optional[`str`]) – The placeholder text that is shown if nothing is selected, if any.
- **min_values** (`int`) – The minimum number of items that must be chosen for this select menu. Defaults to 1 and must be between 0 and 25.
- **max_values** (`int`) – The maximum number of items that must be chosen for this select menu. Defaults to 1 and must be between 1 and 25.
- **options** (List[`discord.SelectOption`]) – A list of options that can be selected in this menu.
- **disabled** (`bool`) – Whether the select is disabled or not.
- **row** (Optional[`int`]) – The relative row this select menu belongs to. A Discord component can only have 5 rows. By default, items are arranged automatically into those 5 rows. If you'd like to control the relative positioning of the row then passing an index is advised. For example, row=1 will show up before row=2. Defaults to `None` , which is automatic ordering. The row number must be between 0 and 4 (i.e. zero indexed).

property **values**

A list of values that have been selected by the user.

Type:
List[`str`]

property **type**

The type of this component.

Type:
`ComponentType`

v: stable

property **options**

A list of options that can be selected in this menu.

A list of options that can be selected in this menu.

Type:
List[`discord.SelectOption`]

add_option (*, *label*, *value*=..., *description*=*None*, *emoji*=*None*, *default*=*False*)

Adds an option to the select menu.

To append a pre-existing `discord.SelectOption` use the `append_option()` method instead.

Parameters:

- **label** (`str`) – The label of the option. This is displayed to users. Can only be up to 100 characters.
- **value** (`str`) – The value of the option. This is not displayed to users. If not given, defaults to the label. Can only be up to 100 characters.
- **description** (Optional[`str`]) – An additional description of the option, if any. Can only be up to 100 characters.
- **emoji** (Optional[Union[`str` , `Emoji` , `PartialEmoji`]]) – The emoji of the option, if available. This can either be a string representing the custom or unicode emoji or an instance of `PartialEmoji` or `Emoji` .
- **default** (`bool`) – Whether this option is selected by default.

Raises:
`ValueError` – The number of options exceeds 25.

append_option (*option*)

Appends an option to the select menu.

Parameters:
option (`discord.SelectOption`) – The option to append to the select menu.

Raises:
`ValueError` – The number of options exceeds 25.

await **callback** (*interaction*)

This function is a *coroutine*.

The callback associated with this UI item.

This can be overridden by subclasses.

Parameters:
interaction (`Interaction`) – The interaction that triggered this UI item.

property **custom_id**

The ID of the select menu that gets received during an interaction.

Type:
str

property **disabled**

Whether the select is disabled or not.

Type:
bool

property **max_values**

The maximum number of items that can be chosen for this select menu.

Type:
int

property **min_values**

The minimum number of items that must be chosen for this select menu.

Type:
int

property **placeholder**

The placeholder text that is shown if nothing is selected, if any.

Type:
Optional[str]

property **view**

The underlying view for this item.

Type:
Optional[View]

ChannelSelect

class discord.ui.**ChannelSelect**(**, custom_id = ..., channel_types = ..., placeholder = None, min_values = 1, max_values = 1, disabled = False, row = None*)

Attributes

channel_types
custom_id
disabled
max_values
min_values

Methods

async callback

v: stable

> placeholder
> type
> values
> view

Represents a UI select menu with a list of predefined options with the current channels in the guild.

Please note that if you use this in a private message with a user, no channels will be displayed to the user.

New in version 2.1.

Parameters:

- **custom_id** (`str`) – The ID of the select menu that gets received during an interaction. If not given then one is generated for you.
- **channel_types** (List[`ChannelType`]) – The types of channels to show in the select menu. Defaults to all channels.
- **placeholder** (Optional[`str`]) – The placeholder text that is shown if nothing is selected, if any.
- **min_values** (`int`) – The minimum number of items that must be chosen for this select menu. Defaults to 1 and must be between 0 and 25.
- **max_values** (`int`) – The maximum number of items that must be chosen for this select menu. Defaults to 1 and must be between 1 and 25.
- **disabled** (`bool`) – Whether the select is disabled or not.
- **row** (Optional[`int`]) – The relative row this select menu belongs to. A Discord component can only have 5 rows. By default, items are arranged automatically into those 5 rows. If you'd like to control the relative positioning of the row then passing an index is advised. For example, row=1 will show up before row=2. Defaults to `None` , which is automatic ordering. The row number must be between 0 and 4 (i.e. zero indexed).

await **callback** (*interaction*)

This function is a *coroutine*.

The callback associated with this UI item.

This can be overridden by subclasses.

Parameters:

interaction (`Interaction`) – The interaction that triggered this UI item.

property **custom_id**

The ID of the select menu that gets received during an interaction. v: stable

Type:

`str`

property **disabled**

Whether the select is disabled or not.

Type:
`bool`

property **max_values**

The maximum number of items that can be chosen for this select menu.

Type:
`int`

property **min_values**

The minimum number of items that must be chosen for this select menu.

Type:
`int`

property **placeholder**

The placeholder text that is shown if nothing is selected, if any.

Type:
Optional[`str`]

property **view**

The underlying view for this item.

Type:
Optional[`View`]

property **type**

The type of this component.

Type:
`ComponentType`

property **channel_types**

A list of channel types that can be selected.

Type:
List[`ChannelType`]

property **values**

A list of channels selected by the user.

Type:
List[Union[`AppCommandChannel` , `AppCommandThread`]]

RoleSelect

class discord.ui.**RoleSelect**(***, *custom_id = ...*, *placeholder = None*, *min_values = 1*, *max_values = 1*, *disabled = False*, *row = None*)

Attributes

custom_id
disabled
max_values
min_values
placeholder
type
values
view

Methods

async callback

Represents a UI select menu with a list of predefined options with the current roles of the guild.

Please note that if you use this in a private message with a user, no roles will be displayed to the user.

New in version 2.1.

Parameters:

- **custom_id** (`str`) – The ID of the select menu that gets received during an interaction. If not given then one is generated for you.
- **placeholder** (Optional[`str`]) – The placeholder text that is shown if nothing is selected, if any.
- **min_values** (`int`) – The minimum number of items that must be chosen for this select menu. Defaults to 1 and must be between 0 and 25.
- **max_values** (`int`) – The maximum number of items that must be chosen for this select menu. Defaults to 1 and must be between 1 and 25.
- **disabled** (`bool`) – Whether the select is disabled or not.
- **row** (Optional[`int`]) – The relative row this select menu belongs to. A Discord component can only have 5 rows. By default, items are arranged automatically into those 5 rows. If you'd like to control the relative positioning of the row then passing an index is advised. For example, row=1 will show up before row=2. Defaults to `None` , which is automatic ordering. The row number must be between 0 and 4 (i.e. zero indexed).

property **type**

The type of this component.

Type:
ComponentType

property **values**

A list of roles that have been selected by the user.

Type:
List[discord.Role]

await **callback** (*interaction*)

This function is a *coroutine*.

The callback associated with this UI item.

This can be overridden by subclasses.

Parameters:
interaction (Interaction) – The interaction that triggered this UI item.

property **custom_id**

The ID of the select menu that gets received during an interaction.

Type:
str

property **disabled**

Whether the select is disabled or not.

Type:
bool

property **max_values**

The maximum number of items that can be chosen for this select menu.

Type:
int

property **min_values**

The minimum number of items that must be chosen for this select menu.

Type:
int

property **placeholder**

The placeholder text that is shown if nothing is selected, if any.

Type:
Optional[str]

property **view**

The underlying view for this item.

Type:
Optional[View]

MentionableSelect

class discord.ui.**MentionableSelect**(***, *custom_id = ...*, *placeholder = None*, *min_values = 1*, *max_values = 1*, *disabled = False*, *row = None*)

Attributes

custom_id
disabled
max_values
min_values
placeholder
type
values
view

Methods

async callback

Represents a UI select menu with a list of predefined options with the current members and roles in the guild.

If this is sent in a private message, it will only allow the user to select the client or themselves. Every selected option in a private message will resolve to a discord.User . It will not give the user any roles to select.

New in version 2.1.

Parameters:

- **custom_id** (str) – The ID of the select menu that gets received during an interaction. If not given then one is generated for you.
- **placeholder** (Optional[str]) – The placeholder text that is shown if nothing is selected, if any.
- **min_values** (int) – The minimum number of items that must be chosen for this select menu. Defaults to 1 and must be between 0 and 25.
- **max_values** (int) – The maximum number of items that must be chosen for this select menu. Defaults to 1 and must be between 1 and 25.
- **disabled** (bool) – Whether the select is disabled or not.
- **row** (Optional[int]) – The relative row this select menu belongs to. A [v: stable component can only have 5 rows. By default, items are arranged automatically into those 5 rows. If you'd like to control the relative positioning of the row then passing an index is advised. For example, row=1 will show up before row=2. Defaults to

an index is advised. For example, row=1 will show up before row=2. Defaults to `None`, which is automatic ordering. The row number must be between 0 and 4 (i.e. zero indexed).

property **type**

The type of this component.

Type:
`ComponentType`

property **values**

A list of roles, members, and users that have been selected by the user.

If this is sent a private message, it will only allow the user to select the client or themselves. Every selected option in a private message will resolve to a `discord.User`.

If invoked in a guild, the values will always resolve to `discord.Member`.

Type:
List[Union[`discord.Role`, `discord.Member`, `discord.User`]]

await **callback**(*interaction*)

This function is a *coroutine*.

The callback associated with this UI item.

This can be overridden by subclasses.

Parameters:
interaction (`Interaction`) – The interaction that triggered this UI item.

property **custom_id**

The ID of the select menu that gets received during an interaction.

Type:
`str`

property **disabled**

Whether the select is disabled or not.

Type:
`bool`

property **max_values**

The maximum number of items that can be chosen for this select menu. v: stable

Type:
`int`

property **min_values**

The minimum number of items that must be chosen for this select menu.

Type:
`int`

property **placeholder**

The placeholder text that is shown if nothing is selected, if any.

Type:
Optional[`str`]

property **view**

The underlying view for this item.

Type:
Optional[`View`]

UserSelect

class discord.ui.**UserSelect**(***, *custom_id = ...*, *placeholder = None*, *min_values = 1*, *max_values = 1*, *disabled = False*, *row = None*)

Attributes

custom_id
disabled
max_values
min_values
placeholder
type
values
view

Methods

async callback

Represents a UI select menu with a list of predefined options with the current members of the guild.

If this is sent a private message, it will only allow the user to select the client or themselves. Every selected option in a private message will resolve to a `discord.User`.

New in version 2.1.

Parameters:

- **custom_id** (`str`) – The ID of the select menu that gets received during an interaction. If not given then one is generated for you.

- **placeholder** (Optional[`str`]) – The placeholder text that is shown if nothing is selected, if any.
- **min_values** (`int`) – The minimum number of items that must be chosen for this select menu. Defaults to 1 and must be between 0 and 25.
- **max_values** (`int`) – The maximum number of items that must be chosen for this select menu. Defaults to 1 and must be between 1 and 25.
- **disabled** (`bool`) – Whether the select is disabled or not.
- **row** (Optional[`int`]) – The relative row this select menu belongs to. A Discord component can only have 5 rows. By default, items are arranged automatically into those 5 rows. If you'd like to control the relative positioning of the row then passing an index is advised. For example, row=1 will show up before row=2. Defaults to `None` , which is automatic ordering. The row number must be between 0 and 4 (i.e. zero indexed).

property **type**

The type of this component.

Type:
`ComponentType`

property **values**

A list of members and users that have been selected by the user.

If this is sent a private message, it will only allow the user to select the client or themselves. Every selected option in a private message will resolve to a `discord.User` .

If invoked in a guild, the values will always resolve to `discord.Member` .

Type:
List[Union[`discord.Member` , `discord.User`]]

await **callback** (*interaction*)

This function is a *coroutine*.

The callback associated with this UI item.

This can be overridden by subclasses.

Parameters:
interaction (`Interaction`) – The interaction that triggered this UI item.

property **custom_id**

The ID of the select menu that gets received during an interaction.

Type:
`str`

property **disabled**

Whether the select is disabled or not.

Type:
bool

property **max_values**

The maximum number of items that can be chosen for this select menu.

Type:
int

property **min_values**

The minimum number of items that must be chosen for this select menu.

Type:
int

property **placeholder**

The placeholder text that is shown if nothing is selected, if any.

Type:
Optional[str]

property **view**

The underlying view for this item.

Type:
Optional[View]

select

@discord.ui.**select**(**, cls = discord.ui.select.Select[+ V], options = ..., channel_types = ..., placeholder = None, custom_id = ..., min_values = 1, max_values = 1, disabled = False, row = None*)

A decorator that attaches a select menu to a component.

The function being decorated should have three parameters, `self` representing the `discord.ui.View`, the `discord.Interaction` you receive and the chosen select class.

To obtain the selected values inside the callback, you can use the `values` attribute of the chosen class in the callback. The list of values will depend on the type of sele v: stable . View the table below for more information.

Select Type	Resolved Values
`discord.ui.Select`	List[`str`]
`discord.ui.UserSelect`	List[Union[`discord.Member` , `discord.User`]]
`discord.ui.RoleSelect`	List[`discord.Role`]
`discord.ui.MentionableSelect`	List[Union[`discord.Role` , `discord.Member` , `discord.User`]]
`discord.ui.ChannelSelect`	List[Union[`AppCommandChannel` , `AppCommandThread`]]

Changed in version 2.1: Added the following keyword-arguments: `cls` , `channel_types`

Example

```
class View(discord.ui.View):

    @discord.ui.select(cls=ChannelSelect, channel_types=[discord.ChannelTy
    async def select_channels(self, interaction: discord.Interaction, sele
        return await interaction.response.send_message(f'You selected {sel
```

Parameters:

- **cls** (Union[Type[`discord.ui.Select`], Type[`discord.ui.UserSelect`], Type[`discord.ui.RoleSelect`], Type[`discord.ui.MentionableSelect`], Type[`discord.ui.ChannelSelect`]]) – The class to use for the select menu. Defaults to `discord.ui.Select` . You can use other select types to display different select menus to the user. See the table above for the different values you can get from each select type. Subclasses work as well, however the callback in the subclass will get overridden.
- **placeholder** (Optional[`str`]) – The placeholder text that is shown if nothing is selected, if any.
- **custom_id** (`str`) – The ID of the select menu that gets received during an interaction. It is recommended not to set this parameter to prevent conflicts.
- **row** (Optional[`int`]) – The relative row this select menu belongs to. A Discord component can only have 5 rows. By default, items are arranged automatically into those 5 rows. If you'd like to control the relative positioning of the row t[illegible] an index is advised. For example, row=1 will show up before row=2. Def `None` , which is automatic ordering. The row number must be between 0 and 4 (i.e. zero indexed).

- **min_values** (`int`) – The minimum number of items that must be chosen for this select menu. Defaults to 1 and must be between 0 and 25.
- **max_values** (`int`) – The maximum number of items that must be chosen for this select menu. Defaults to 1 and must be between 1 and 25.
- **options** (List[`discord.SelectOption`]) – A list of options that can be selected in this menu. This can only be used with `Select` instances.
- **channel_types** (List[`ChannelType`]) – The types of channels to show in the select menu. Defaults to all channels. This can only be used with `ChannelSelect` instances.
- **disabled** (`bool`) – Whether the select is disabled or not. Defaults to `False` .

TextInput

class discord.ui.**TextInput**(***, *label*, *style=<TextStyle.short: 1>*, *custom_id=...*, *placeholder=None*, *default=None*, *required=True*, *min_length=None*, *max_length=None*, *row=None*)

Attributes

custom_id
default
label
max_length
min_length
placeholder
required
style
value
view

Methods

async callback

Represents a UI text input.

Supported Operations

`str(x)`

Returns the value of the text input or an empty string if the value is `None` .

New in version 2.0.

Parameters:

- **label** (`str`) – The label to display above the text input.
- **custom_id** (`str`) – The ID of the text input that gets received during an interaction. If not given then one is generated for you.
- **style** (`discord.TextStyle`) – The style of the text input.
- **placeholder** (Optional[`str`]) – The placeholder text to display when the text input is

empty.

- **default** (Optional[`str`]) – The default value of the text input.
- **required** (`bool`) – Whether the text input is required.
- **min_length** (Optional[`int`]) – The minimum length of the text input.
- **max_length** (Optional[`int`]) – The maximum length of the text input.
- **row** (Optional[`int`]) – The relative row this text input belongs to. A Discord component can only have 5 rows. By default, items are arranged automatically into those 5 rows. If you'd like to control the relative positioning of the row then passing an index is advised. For example, row=1 will show up before row=2. Defaults to `None` , which is automatic ordering. The row number must be between 0 and 4 (i.e. zero indexed).

property **custom_id**

The ID of the text input that gets received during an interaction.

Type:
`str`

property **value**

The value of the text input.

Type:
`str`

property **label**

The label of the text input.

Type:
`str`

property **placeholder**

The placeholder text to display when the text input is empty.

Type:
`str`

property **required**

Whether the text input is required.

Type:
`bool`

property **min_length**

The minimum length of the text input.

Type:
`int`

property **max_length**

The maximum length of the text input.

Type:
`int`

property **style**

The style of the text input.

Type:
`discord.TextStyle`

await **callback**(*interaction*)

This function is a *coroutine*.

The callback associated with this UI item.

This can be overridden by subclasses.

Parameters:
interaction (`Interaction`) – The interaction that triggered this UI item.

property **view**

The underlying view for this item.

Type:
Optional[`View`]

property **default**

The default value of the text input.

Type:
`str`

Application Commands

The library has helpers to aid in creation of application commands. These are all in the `discord.app_commands` package.

CommandTree

v: stable

class discord.app_commands. **CommandTree**(*client*, *, *fallback_to_global* = *True*)

Attributes

translator

Methods

def add_command
def clear_commands
@ command
@ context_menu
def copy_global_to
@ error
async fetch_command
async fetch_commands
def get_command
def get_commands
async interaction_check
async on_error
def remove_command
async set_translator
async sync
def walk_commands

Represents a container that holds application command information.

Parameters:

- **client** (`Client`) – The client instance to get application command information from.
- **fallback_to_global** (`bool`) – If a guild-specific command is not found when invoked, then try falling back into a global command in the tree. For example, if the tree locally has a `/ping` command under the global namespace but the guild has a guild-specific `/ping` , instead of failing to find the guild-specific `/ping` command it will fall back to the global `/ping` command. This has the potential to raise more `CommandSignatureMismatch` errors than usual. Defaults to `True` .

@ **command** (`*` , *name* = `...` , *description* = `...` , *nsfw* = *False* , *guild* = `...` , *guilds* = `...` , *auto_locale_strings* = *True* , *extras* = `...`)

A decorator that creates an application command from a regular function directly under this tree.

Parameters:

- **name** (Union[`str` , `locale_str`]) – The name of the application command. If not given, it defaults to a lower-case version of the callback name.
- **description** (Union[`str` , `locale_str`]) – The description of the application command. This shows up in the UI to describe the application command. If not given, it defaults to the first line of the docstring of the callback shortened to 100 characters.
- **nsfw** (`bool`) –
  Whether the command is NSFW and should only work in NSFW channels. Defaults to `False` .

Due to a Discord limitation, this does not work on subcommands.

- **guild** (Optional[`Snowflake`]) – The guild to add the command to. If not given or `None` then it becomes a global command instead.
- **guilds** (List[`Snowflake`]) – The list of guilds to add the command to. This cannot be mixed with the `guild` parameter. If no guilds are given at all then it becomes a global command instead.
- **auto_locale_strings** (`bool`) – If this is set to `True` , then all translatable strings will implicitly be wrapped into `locale_str` rather than `str` . This could avoid some repetition and be more ergonomic for certain defaults such as default command names, command descriptions, and parameter names. Defaults to `True` .
- **extras** (`dict`) – A dictionary that can be used to store extraneous data. The library will not touch any values or keys within this dictionary.

@ **context_menu** (*, *name* = ..., *nsfw* = *False*, *guild* = ..., *guilds* = ..., *auto_locale_strings* = *True*, *extras* = ...)

A decorator that creates an application command context menu from a regular function directly under this tree.

This function must have a signature of `Interaction` as its first parameter and taking either a `Member` , `User` , or `Message` , or a `typing.Union` of `Member` and `User` as its second parameter.

Examples

```
@app_commands.context_menu()
async def react(interaction: discord.Interaction, message: discord.Mess
    await interaction.response.send_message('Very cool message!', ephem

@app_commands.context_menu()
async def ban(interaction: discord.Interaction, user: discord.Member):
    await interaction.response.send_message(f'Should I actually ban {us
```

Parameters:

- **name** (Union[`str` , `locale_str`]) – The name of the context menu command. If not given, it defaults to a title-case version of the callback name. Note that unlike regular slash commands this can have spaces and upper case characters in the name.
- **nsfw** (`bool`) –
  Whether the command is NSFW and should only work in NSFW channels. Defaults to `False` .
  Due to a Discord limitation, this does not work on subcommands.
- **guild** (Optional[`Snowflake`]) – The guild to add the command to. If not given or `None` then it becomes a global command instead.

- **guilds** (List[Snowflake]) – The list of guilds to add the command to. This cannot be mixed with the `guild` parameter. If no guilds are given at all then it becomes a global command instead.
- **auto_locale_strings** (`bool`) – If this is set to `True` , then all translatable strings will implicitly be wrapped into `locale_str` rather than `str` . This could avoid some repetition and be more ergonomic for certain defaults such as default command names, command descriptions, and parameter names. Defaults to `True` .
- **extras** (`dict`) – A dictionary that can be used to store extraneous data. The library will not touch any values or keys within this dictionary.

@**error**(*coro*)

A decorator that registers a coroutine as a local error handler.

This must match the signature of the `on_error()` callback.

The error passed will be derived from `AppCommandError` .

Parameters:

coro (coroutine) – The coroutine to register as the local error handler.

Raises:

`TypeError` – The coroutine passed is not actually a coroutine or does not match the signature.

await **fetch_command**(*command_id* , */* , ***** , *guild=None*)

This function is a *coroutine*.

Fetches an application command from the application.

Parameters:

- **command_id** (`int`) – The ID of the command to fetch.
- **guild** (Optional[Snowflake]) – The guild to fetch the command from. If not passed then the global command is fetched instead.

Raises:

- `HTTPException` – Fetching the command failed.
- `MissingApplicationID` – The application ID could not be found.
- `NotFound` – The application command was not found. This could also be because the command is a guild command and the guild was not specified and vice versa.

Returns:

The application command.

v: stable

Return type:

`AppCommand`

await **fetch_commands**(**, guild=None*)

This function is a *coroutine*.

Fetches the application's current commands.

If no guild is passed then global commands are fetched, otherwise the guild's commands are fetched instead.

> **Note**
>
> This includes context menu commands.

Parameters:

guild (Optional[Snowflake]) – The guild to fetch the commands from. If not passed then global commands are fetched instead.

Raises:

- HTTPException – Fetching the commands failed.
- MissingApplicationID – The application ID could not be found.

Returns:

The application's commands.

Return type:

List[AppCommand]

copy_global_to(**, guild*)

Copies all global commands to the specified guild.

This method is mainly available for development purposes, as it allows you to copy your global commands over to a testing guild easily.

Note that this method will *override* pre-existing guild commands that would conflict.

Parameters:

guild (Snowflake) – The guild to copy the commands to.

Raises:

CommandLimitReached – The maximum number of commands was reached for that guild. This is currently 100 for slash commands and 5 for context menu commands.

add_command(*command, /, *, guild=..., guilds=..., override=False*)

v: stable

Adds an application command to the tree.

This only adds the command locally – in order to sync the commands and enable them

in the client, `sync()` must be called.

The root parent of the command is added regardless of the type passed.

Parameters:

- **command** (Union[`Command` , `Group`]) – The application command or group to add.
- **guild** (Optional[`Snowflake`]) – The guild to add the command to. If not given or `None` then it becomes a global command instead.
- **guilds** (List[`Snowflake`]) – The list of guilds to add the command to. This cannot be mixed with the `guild` parameter. If no guilds are given at all then it becomes a global command instead.
- **override** (`bool`) – Whether to override a command with the same name. If `False` an exception is raised. Default is `False` .

Raises:

- `CommandAlreadyRegistered` – The command was already registered and no override was specified.
- `TypeError` – The application command passed is not a valid application command. Or, `guild` and `guilds` were both given.
- `CommandLimitReached` – The maximum number of commands was reached globally or for that guild. This is currently 100 for slash commands and 5 for context menu commands.

remove_command (*command* , */* , *** , *guild=None* , *type=<AppCommandType.chat_input: 1>*)

Removes an application command from the tree.

This only removes the command locally – in order to sync the commands and remove them in the client, `sync()` must be called.

Parameters:

- **command** (`str`) – The name of the root command to remove.
- **guild** (Optional[`Snowflake`]) – The guild to remove the command from. If not given or `None` then it removes a global command instead.
- **type** (`AppCommandType`) – The type of command to remove. Defaults to `chat_input` , i.e. slash commands.

Returns:

The application command that got removed. If nothing was removed then `None` is returned instead.

Return type:

Optional[Union[`Command` , `ContextMenu` , `Group`]]

v: stable

clear_commands (*** , *guild* , *type = None*)

Clears all application commands from the tree.

This only removes the commands locally – in order to sync the commands and remove them in the client, `sync()` must be called.

Parameters:

- **guild** (Optional[`Snowflake`]) – The guild to remove the commands from. If `None` then it removes all global commands instead.
- **type** (`AppCommandType`) – The type of command to clear. If not given or `None` then it removes all commands regardless of the type.

get_command (*command*, */*, ***, *guild=None*, *type=<AppCommandType.chat_input: 1>*)

Gets an application command from the tree.

Parameters:

- **command** (`str`) – The name of the root command to get.
- **guild** (Optional[`Snowflake`]) – The guild to get the command from. If not given or `None` then it gets a global command instead.
- **type** (`AppCommandType`) – The type of command to get. Defaults to `chat_input` , i.e. slash commands.

Returns:

The application command that was found. If nothing was found then `None` is returned instead.

Return type:

Optional[Union[`Command` , `ContextMenu` , `Group`]]

get_commands (***, *guild = None* , *type = None*)

Gets all application commands from the tree.

Parameters:

- **guild** (Optional[`Snowflake`]) – The guild to get the commands from, not including global commands. If not given or `None` then only global commands are returned.
- **type** (Optional[`AppCommandType`]) – The type of commands to get. When not given or `None` , then all command types are returned.

Returns:

The application commands from the tree.

Return type:

List[Union[`ContextMenu` , `Command` , `Group`]]

for ... in **walk_commands** (***, *guild=None*, *type=<AppCommandType.chat_input: 1>*)

An iterator that recursively walks through all application commands and child commands from the tree.

Parameters:

- **guild** (Optional[`Snowflake`]) – The guild to iterate the commands from, not including global commands. If not given or `None` then only global commands are iterated.
- **type** (`AppCommandType`) – The type of commands to iterate over. Defaults to `chat_input` , i.e. slash commands.

Yields:

Union[`ContextMenu` , `Command` , `Group`] – The application commands from the tree.

await **on_error**(*interaction* , *error* , */*)

This function is a *coroutine*.

A callback that is called when any command raises an `AppCommandError` .

The default implementation logs the exception using the library logger if the command does not have any error handlers attached to it.

To get the command that failed, `discord.Interaction.command` should be used.

Parameters:

- **interaction** (`Interaction`) – The interaction that is being handled.
- **error** (`AppCommandError`) – The exception that was raised.

property **translator**

The translator, if any, responsible for handling translation of commands.

To change the translator, use `set_translator()` .

Type:

Optional[`Translator`]

await **set_translator**(*translator*)

This function is a *coroutine*.

Sets the translator to use for translating commands.

If a translator was previously set, it will be unloaded using its `Translator.unload()` method.

When a translator is set, it will be loaded using its `Translator.load()` m v: stable

Parameters:

translator (Optional[`Translator`]) – The translator to use. If `None` then the

translator is just removed and unloaded.

Raises:
`TypeError` – The translator was not `None` or a `Translator` instance.

await **sync**(**, guild=None*)

This function is a *coroutine*.

Syncs the application commands to Discord.

This also runs the translator to get the translated strings necessary for feeding back into Discord.

This must be called for the application commands to show up.

Parameters:
guild (Optional[`Snowflake`]) – The guild to sync the commands to. If `None` then it syncs all global commands instead.

Raises:
- `HTTPException` – Syncing the commands failed.
- `CommandSyncFailure` – Syncing the commands failed due to a user related error, typically because the command has invalid data. This is equivalent to an HTTP status code of 400.
- `Forbidden` – The client does not have the `applications.commands` scope in the guild.
- `MissingApplicationID` – The client does not have an application ID.
- `TranslationError` – An error occurred while translating the commands.

Returns:
The application's commands that got synced.

Return type:
List[`AppCommand`]

await **interaction_check**(*interaction, /*)

This function is a *coroutine*.

A global check to determine if an `Interaction` should be processed by the tree.

The default implementation returns True (all interactions are processed), but can be overridden if custom behaviour is desired.

Commands

v: stable

Command

Command

class discord.app_commands. **Command** (***, *name*, *description*, *callback*, *nsfw=False*, *parent=None*, *guild_ids=None*, *auto_locale_strings=True*, *extras=...*)

Attributes

callback
checks
default_permissions
description
extras
guild_only
name
nsfw
parameters
parent
qualified_name
root_parent

Methods

def add_check
@ autocomplete
@ error
def get_parameter
def remove_check

A class that implements an application command.

These are usually not created manually, instead they are created using one of the following decorators:

- `command()`
- `Group.command`
- `CommandTree.command`

New in version 2.0.

Parameters:

- **name** (Union[`str` , `locale_str`]) – The name of the application command.
- **description** (Union[`str` , `locale_str`]) – The description of the application command. This shows up in the UI to describe the application command.
- **callback** (coroutine) – The coroutine that is executed when the command is called.
- **auto_locale_strings** (`bool`) – If this is set to `True` , then all translatable strings will implicitly be wrapped into `locale_str` rather than `str` . This could avoid some repetition and be more ergonomic for certain defaults such as default command names, command descriptions, and parameter names. Defaults to `True` .
- **nsfw** (`bool`) –
  Whether the command is NSFW and should only work in NSFW channels. Defaults to `False` .
  Due to a Discord limitation, this does not work on subcommands.
- **parent** (Optional[`Group`]) – The parent application command. `None` if there isn't one.
- **extras** (`dict`) – A dictionary that can be used to store extraneous data. The library

- **extras** (dict) – A dictionary that can be used to store extraneous data. The library will not touch any values or keys within this dictionary.

name

The name of the application command.

Type:
str

description

The description of the application command. This shows up in the UI to describe the application command.

Type:
str

checks

A list of predicates that take a Interaction parameter to indicate whether the command callback should be executed. If an exception is necessary to be thrown to signal failure, then one inherited from AppCommandError should be used. If all the checks fail without propagating an exception, CheckFailure is raised.

default_permissions

The default permissions that can execute this command on Discord. Note that server administrators can override this value in the client. Setting an empty permissions field will disallow anyone except server administrators from using the command in a guild.

Due to a Discord limitation, this does not work on subcommands.

Type:
Optional[Permissions]

guild_only

Whether the command should only be usable in guild contexts.

Due to a Discord limitation, this does not work on subcommands.

Type:
bool

nsfw

Whether the command is NSFW and should only work in NSFW channels.

Due to a Discord limitation, this does not work on subcommands.

Type:
bool

parent

parent

The parent application command. `None` if there isn't one.

Type:
Optional[`Group`]

extras

A dictionary that can be used to store extraneous data. The library will not touch any values or keys within this dictionary.

Type:
`dict`

@autocomplete(*name*)

A decorator that registers a coroutine as an autocomplete prompt for a parameter.

The coroutine callback must have 2 parameters, the `Interaction` , and the current value by the user (the string currently being typed by the user).

To get the values from other parameters that may be filled in, accessing `Interaction.namespace` will give a `Namespace` object with those values.

Parent `checks` are ignored within an autocomplete. However, checks can be added to the autocomplete callback and the ones added will be called. If the checks fail for any reason then an empty list is sent as the interaction response.

The coroutine decorator **must** return a list of `Choice` objects. Only up to 25 objects are supported.

> **Warning**
>
> The choices returned from this coroutine are suggestions. The user may ignore them and input their own value.

Example:

```
@app_commands.command()
async def fruits(interaction: discord.Interaction, fruit: str):
    await interaction.response.send_message(f'Your favourite fruit seem

@fruits.autocomplete('fruit')
async def fruits_autocomplete(
    interaction: discord.Interaction,
    current: str,
) -> List[app_commands.Choice[str]]:
    fruits = ['Banana', 'Pineapple', 'Apple', 'Watermelon', 'Melon', 'C
    return [
        app_commands.Choice(name=fruit, value=fruit)
```

v: stable

```
        for fruit in fruits if current.lower() in fruit.lower()
    ]
```

Parameters:

name (`str`) – The parameter name to register as autocomplete.

Raises:

`TypeError` – The coroutine passed is not actually a coroutine or the parameter is not found or of an invalid type.

@**error** (*coro*)

A decorator that registers a coroutine as a local error handler.

The local error handler is called whenever an exception is raised in the body of the command or during handling of the command. The error handler must take 2 parameters, the interaction and the error.

The error passed will be derived from `AppCommandError` .

Parameters:

coro (coroutine) – The coroutine to register as the local error handler.

Raises:

`TypeError` – The coroutine passed is not actually a coroutine.

property **callback**

The coroutine that is executed when the command is called.

Type:

coroutine

property **parameters**

Returns a list of parameters for this command.

This does not include the `self` or `interaction` parameters.

Returns:

The parameters of this command.

Return type:

List[`Parameter`]

get_parameter (*name*)

Retrieves a parameter by its name.

The name must be the Python identifier rather than the renamed one for d v: stable
Discord.

Parameters:

name (`str`) – The parameter name in the callback function.

Returns:

The parameter or `None` if not found.

Return type:

Optional[`Parameter`]

property **root_parent**

The root parent of this command.

Type:

Optional[`Group`]

property **qualified_name**

Returns the fully qualified command name.

The qualified name includes the parent name as well. For example, in a command like `/foo bar` the qualified name is `foo bar` .

Type:

`str`

add_check (*func* , */*)

Adds a check to the command.

This is the non-decorator interface to `check()` .

Parameters:

func – The function that will be used as a check.

remove_check (*func* , */*)

Removes a check from the command.

This function is idempotent and will not raise an exception if the function is not in the command's checks.

Parameters:

func – The function to remove from the checks.

Parameter

class discord.app_commands. **Parameter**

v: stable

Attributes

> autocomplete

channel_types
choices
command
default
description
display_name
locale_description
locale_name
max_value
min_value
name
required
type

A class that contains the parameter information of a `Command` callback.

New in version 2.0.

name

The name of the parameter. This is the Python identifier for the parameter.

Type:
`str`

display_name

The displayed name of the parameter on Discord.

Type:
`str`

description

The description of the parameter.

Type:
`str`

autocomplete

Whether the parameter has an autocomplete handler.

Type:
`bool`

locale_name

The display name's locale string, if available.

Type:
Optional[`locale_str`]

locale_description

The description's locale string, if available.

Type:
Optional[`locale_str`]

required

Whether the parameter is required

Type:
`bool`

choices

A list of choices this parameter takes, if any.

Type:
List[`Choice`]

type

The underlying type of this parameter.

Type:
`AppCommandOptionType`

channel_types

The channel types that are allowed for this parameter.

Type:
List[`ChannelType`]

min_value

The minimum supported value for this parameter.

Type:
Optional[Union[`int` , `float`]]

max_value

The maximum supported value for this parameter.

Type:
Optional[Union[`int` , `float`]]

default

The default value of the parameter, if given. If not given then this is `MISSING` .

Type:
Any

command

The command this parameter is attached to.

Type:

Command

ContextMenu

class discord.app_commands. **ContextMenu** (**, name, callback, type = ..., nsfw = False, guild_ids = None, auto_locale_strings = True, extras = ...*)

Attributes

callback
checks
default_permissions
extras
guild_only
name
nsfw
qualified_name
type

Methods

def add_check
@ error
def remove_check

A class that implements a context menu application command.

These are usually not created manually, instead they are created using one of the following decorators:

- context_menu()
- CommandTree.context_menu

New in version 2.0.

Parameters:

- **name** (Union[str , locale_str]) – The name of the context menu.
- **callback** (coroutine) – The coroutine that is executed when the command is called.
- **type** (AppCommandType) – The type of context menu application command. By default, this is inferred by the parameter of the callback.
- **auto_locale_strings** (bool) – If this is set to True , then all translatable strings will implicitly be wrapped into locale_str rather than str . This could avoid some repetition and be more ergonomic for certain defaults such as default command names, command descriptions, and parameter names. Defaults to True .
- **nsfw** (bool) – Whether the command is NSFW and should only work in NSFW channels. Defaults to False .
- **extras** (dict) – A dictionary that can be used to store extraneous data. The library will not touch any values or keys within this dictionary.

name

The name of the context menu.

Type:

str

type

The type of context menu application command. By default, this is inferred by the parameter of the callback.

Type:

AppCommandType

default_permissions

The default permissions that can execute this command on Discord. Note that server administrators can override this value in the client. Setting an empty permissions field will disallow anyone except server administrators from using the command in a guild.

Type:

Optional[Permissions]

guild_only

Whether the command should only be usable in guild contexts. Defaults to `False`.

Type:

bool

nsfw

Whether the command is NSFW and should only work in NSFW channels. Defaults to `False`.

Type:

bool

checks

A list of predicates that take a `Interaction` parameter to indicate whether the command callback should be executed. If an exception is necessary to be thrown to signal failure, then one inherited from `AppCommandError` should be used. If all the checks fail without propagating an exception, `CheckFailure` is raised.

extras

A dictionary that can be used to store extraneous data. The library will not touch any values or keys within this dictionary.

Type:

dict

@**error**(*coro*)

A decorator that registers a coroutine as a local error handler.

The local error handler is called whenever an exception is raised in the body of the command or during handling of the command. The error handler must take 2 parameters, the interaction and the error.

The error passed will be derived from `AppCommandError`.

Parameters:

coro (coroutine) – The coroutine to register as the local error handler.

Raises:

`TypeError` – The coroutine passed is not actually a coroutine.

property **callback**

The coroutine that is executed when the context menu is called.

Type:

coroutine

property **qualified_name**

Returns the fully qualified command name.

Type:

`str`

add_check(*func*, */*)

Adds a check to the command.

This is the non-decorator interface to `check()`.

Parameters:

func – The function that will be used as a check.

remove_check(*func*, */*)

Removes a check from the command.

This function is idempotent and will not raise an exception if the function is not in the command's checks.

Parameters:

func – The function to remove from the checks.

Group

class discord.app_commands.**Group**(***, *name=...*, *description=...*, *parent=None*, *guild_ids=None*, *guild_only=...*, *nsfw=...*, *auto_locale_strings=True*, *default_permissions=...*, *extras=...*)

Attributes

commands
default_permissions
description
extras
guild_only
name
nsfw
parent
qualified_name
root_parent

Methods

def add_command
@ command
@ error
def get_command
async interaction_check
async on_error
def remove_command
def walk_commands

A class that implements an application command group.

These are usually inherited rather than created manually.

Decorators such as `guild_only()`, `guilds()`, and `default_permissions()` will apply to the group if used on top of a subclass. For example:

```
from discord import app_commands

@app_commands.guild_only()
class MyGroup(app_commands.Group):
    pass
```

New in version 2.0.

Parameters:

- **name** (Union[`str`, `locale_str`]) – The name of the group. If not given, it defaults to a lower-case kebab-case version of the class name.
- **description** (Union[`str`, `locale_str`]) – The description of the group. This shows up in the UI to describe the group. If not given, it defaults to the docstring of the class shortened to 100 characters.
- **auto_locale_strings** (`bool`) – If this is set to `True`, then all translatable strings will implicitly be wrapped into `locale_str` rather than `str`. This could avoid some repetition and be more ergonomic for certain defaults such as default command names, command descriptions, and parameter names. Defaults to `True`.
- **default_permissions** (Optional[`Permissions`]) –
  The default permissions that can execute this group on Discord. Note that server administrators can override this value in the client. Setting an empty pe v: stable field will disallow anyone except server administrators from using the command in a guild.
  Due to a Discord limitation, this does not work on subcommands

Due to a Discord limitation, this does not work on subcommands.

- **guild_only** (`bool`) –
  Whether the group should only be usable in guild contexts. Defaults to `False` .
  Due to a Discord limitation, this does not work on subcommands.
- **nsfw** (`bool`) –
  Whether the command is NSFW and should only work in NSFW channels. Defaults to `False` .
  Due to a Discord limitation, this does not work on subcommands.
- **parent** (Optional[`Group`]) – The parent application command. `None` if there isn't one.
- **extras** (`dict`) – A dictionary that can be used to store extraneous data. The library will not touch any values or keys within this dictionary.

name

The name of the group.

Type:
`str`

description

The description of the group. This shows up in the UI to describe the group.

Type:
`str`

default_permissions

The default permissions that can execute this group on Discord. Note that server administrators can override this value in the client. Setting an empty permissions field will disallow anyone except server administrators from using the command in a guild.

Due to a Discord limitation, this does not work on subcommands.

Type:
Optional[`Permissions`]

guild_only

Whether the group should only be usable in guild contexts.

Due to a Discord limitation, this does not work on subcommands.

Type:
`bool`

nsfw

Whether the command is NSFW and should only work in NSFW channels. v: stable

Due to a Discord limitation, this does not work on subcommands.

Type:
bool

parent

The parent group. None if there isn't one.

Type:
Optional[Group]

extras

A dictionary that can be used to store extraneous data. The library will not touch any values or keys within this dictionary.

Type:
dict

@ command (*, *name* = ..., *description* = ..., *nsfw* = *False*, *auto_locale_strings* = *True*, *extras* = ...)

A decorator that creates an application command from a regular function under this group.

Parameters:

- **name** (Union[str , locale_str]) – The name of the application command. If not given, it defaults to a lower-case version of the callback name.
- **description** (Union[str , locale_str]) – The description of the application command. This shows up in the UI to describe the application command. If not given, it defaults to the first line of the docstring of the callback shortened to 100 characters.
- **nsfw** (bool) – Whether the command is NSFW and should only work in NSFW channels. Defaults to False .
- **auto_locale_strings** (bool) – If this is set to True , then all translatable strings will implicitly be wrapped into locale_str rather than str . This could avoid some repetition and be more ergonomic for certain defaults such as default command names, command descriptions, and parameter names. Defaults to True .
- **extras** (dict) – A dictionary that can be used to store extraneous data. The library will not touch any values or keys within this dictionary.

@ error (*coro*)

A decorator that registers a coroutine as a local error handler.

The local error handler is called whenever an exception is raised in a child command. The error handler must take 2 parameters, the interaction and the error. v: stable

The error passed will be derived from AppCommandError .

Parameters:

coro (coroutine) – The coroutine to register as the local error handler.

Raises:

`TypeError` – The coroutine passed is not actually a coroutine, or is an invalid coroutine.

property **root_parent**

The parent of this group.

Type:

Optional[`Group`]

property **qualified_name**

Returns the fully qualified group name.

The qualified name includes the parent name as well. For example, in a group like `/foo bar` the qualified name is `foo bar`.

Type:

`str`

property **commands**

The commands that this group contains.

Type:

List[Union[`Command` , `Group`]]

for ... in **walk_commands**()

An iterator that recursively walks through all commands that this group contains.

Yields:

Union[`Command` , `Group`] – The commands in this group.

await **on_error**(*interaction*, *error*, */*)

This function is a *coroutine*.

A callback that is called when a child's command raises an `AppCommandError`.

To get the command that failed, `discord.Interaction.command` should be used.

The default implementation does nothing.

Parameters:

- **interaction** (`Interaction`) – The interaction that is being handled.
- **error** (`AppCommandError`) – The exception that was raised.

await **interaction_check**(*interaction*, */*)

This function is a *coroutine*.

A callback that is called when an interaction happens within the group that checks whether a command inside the group should be executed.

This is useful to override if, for example, you want to ensure that the interaction author is a given user.

The default implementation of this returns `True`.

> **Note**
>
> If an exception occurs within the body then the check is considered a failure and error handlers such as `on_error()` is called. See `AppCommandError` for more information.

Parameters:

interaction (`Interaction`) – The interaction that occurred.

Returns:

Whether the view children's callbacks should be called.

Return type:

`bool`

add_command(*command*, */*, ***, *override=False*)

Adds a command or group to this group's internal list of commands.

Parameters:

- **command** (Union[`Command`, `Group`]) – The command or group to add.
- **override** (`bool`) – Whether to override a pre-existing command or group with the same name. If `False` then an exception is raised.

Raises:

- `CommandAlreadyRegistered` – The command or group is already registered. Note that the `CommandAlreadyRegistered.guild_id` attribute will always be `None` in this case.
- `ValueError` – There are too many commands already registered or the group is too deeply nested.
- `TypeError` – The wrong command type was passed.

remove_command(*name*, */*)

Removes a command or group from the internal list of commands.

Parameters:

name (`str`) – The name of the command or group to remove.

Returns:

The command that was removed. If nothing was removed then `None` is returned instead.

Return type:
Optional[Union[`Command` , `Group`]]

`get_command` (*name* , */*)

Retrieves a command or group from its name.

Parameters:
name (`str`) – The name of the command or group to retrieve.

Returns:
The command or group that was retrieved. If nothing was found then `None` is returned instead.

Return type:
Optional[Union[`Command` , `Group`]]

Decorators

`@discord.app_commands.` `command` (`*` , *name* `=...` , *description* `=...` , *nsfw* `=False` , *auto_locale_strings* `=True` , *extras* `=...`)

Creates an application command from a regular function.

Parameters:

- **name** (`str`) – The name of the application command. If not given, it defaults to a lower-case version of the callback name.
- **description** (`str`) – The description of the application command. This shows up in the UI to describe the application command. If not given, it defaults to the first line of the docstring of the callback shortened to 100 characters.
- **nsfw** (`bool`) –
  Whether the command is NSFW and should only work in NSFW channels. Defaults to `False` .
  Due to a Discord limitation, this does not work on subcommands.
- **auto_locale_strings** (`bool`) – If this is set to `True` , then all translatable strings will implicitly be wrapped into `locale_str` rather than `str` . This could avoid some repetition and be more ergonomic for certain defaults such as default command names, command descriptions, and parameter names. Defaults to `True` .
- **extras** (`dict`) – A dictionary that can be used to store extraneous data. The library will not touch any values or keys within this dictionary.

`@discord.app_commands.` `context_menu` (`*` , *name* `=...` , *nsfw* `=False` , *auto_locale_strings* `=True` , *extras* `=...`)

Creates an application command context menu from a regular function.

This function must have a signature of `Interaction` as its first parameter and taking either a `Member`, `User`, or `Message`, or a `typing.Union` of `Member` and `User` as its second parameter.

Examples

```
@app_commands.context_menu()
async def react(interaction: discord.Interaction, message: discord.Message
    await interaction.response.send_message('Very cool message!', ephemera

@app_commands.context_menu()
async def ban(interaction: discord.Interaction, user: discord.Member):
    await interaction.response.send_message(f'Should I actually ban {user}
```

Parameters:

- **name** (Union[`str`, `locale_str`]) – The name of the context menu command. If not given, it defaults to a title-case version of the callback name. Note that unlike regular slash commands this can have spaces and upper case characters in the name.
- **nsfw** (`bool`) –
  Whether the command is NSFW and should only work in NSFW channels. Defaults to `False`.
  Due to a Discord limitation, this does not work on subcommands.
- **auto_locale_strings** (`bool`) – If this is set to `True`, then all translatable strings will implicitly be wrapped into `locale_str` rather than `str`. This could avoid some repetition and be more ergonomic for certain defaults such as default command names, command descriptions, and parameter names. Defaults to `True`.
- **extras** (`dict`) – A dictionary that can be used to store extraneous data. The library will not touch any values or keys within this dictionary.

@discord.app_commands.**describe**(*******parameters*)

Describes the given parameters by their name using the key of the keyword argument as the name.

Example:

```
@app_commands.command(description='Bans a member')
@app_commands.describe(member='the member to ban')
async def ban(interaction: discord.Interaction, member: discord.Member):
    await interaction.response.send_message(f'Banned {member}')
```

v: stable

Alternatively, you can describe parameters using Google, Sphinx, or Numpy style docstrings.

Example:

```
@app_commands.command()
async def ban(interaction: discord.Interaction, member: discord.Member):
    """Bans a member

    Parameters
    -----------
    member: discord.Member
        the member to ban
    """
    await interaction.response.send_message(f'Banned {member}')
```

Parameters:

****parameters** (Union[`str` , `locale_str`]) – The description of the parameters.

Raises:

`TypeError` – The parameter name is not found.

@discord.app_commands. **rename** (** *parameters*)

Renames the given parameters by their name using the key of the keyword argument as the name.

This renames the parameter within the Discord UI. When referring to the parameter in other decorators, the parameter name used in the function is used instead of the renamed one.

Example:

```
@app_commands.command()
@app_commands.rename(the_member_to_ban='member')
async def ban(interaction: discord.Interaction, the_member_to_ban: discorc
    await interaction.response.send_message(f'Banned {the_member_to_ban}')
```

Parameters:

****parameters** (Union[`str` , `locale_str`]) – The name of the parameters.

Raises:

- `ValueError` – The parameter name is already used by another parameter.
- `TypeError` – The parameter name is not found.

@discord.app_commands. **choices** (** *parameters*)

Instructs the given parameters by their name to use the given choices for their choices.

Example:


```
@app_commands.command()
```

```
@app_commands.describe(fruits='fruits to choose from')
@app_commands.choices(fruits=[
    Choice(name='apple', value=1),
    Choice(name='banana', value=2),
    Choice(name='cherry', value=3),
])
async def fruit(interaction: discord.Interaction, fruits: Choice[int]):
    await interaction.response.send_message(f'Your favourite fruit is {fru
```

Note

This is not the only way to provide choices to a command. There are two more ergonomic ways of doing this. The first one is to use a `typing.Literal` annotation:

```
@app_commands.command()
@app_commands.describe(fruits='fruits to choose from')
async def fruit(interaction: discord.Interaction, fruits: Literal['app
    await interaction.response.send_message(f'Your favourite fruit is
```

The second way is to use an `enum.Enum`:

```
class Fruits(enum.Enum):
    apple = 1
    banana = 2
    cherry = 3

@app_commands.command()
@app_commands.describe(fruits='fruits to choose from')
async def fruit(interaction: discord.Interaction, fruits: Fruits):
    await interaction.response.send_message(f'Your favourite fruit is
```

Parameters:

****parameters** – The choices of the parameters.

Raises:

`TypeError` – The parameter name is not found or the parameter type was incorrect.

@discord.app_commands.**autocomplete**(***parameters*)

Associates the given parameters with the given autocomplete callback.

Autocomplete is only supported on types that have `str`, `int`, or `float` values.

`Checks` are supported, however they must be attached to the autocomplete (v: stable ▾ order to work. Checks attached to the command are ignored when invoking the autocomplete callback.

For more information, see the `Command.autocomplete()` documentation.

> **Warning**
>
> The choices returned from this coroutine are suggestions. The user may ignore them and input their own value.

Example:

```
async def fruit_autocomplete(
    interaction: discord.Interaction,
    current: str,
) -> List[app_commands.Choice[str]]:
    fruits = ['Banana', 'Pineapple', 'Apple', 'Watermelon', 'Melon', 'Cher
    return [
        app_commands.Choice(name=fruit, value=fruit)
        for fruit in fruits if current.lower() in fruit.lower()
    ]

@app_commands.command()
@app_commands.autocomplete(fruit=fruit_autocomplete)
async def fruits(interaction: discord.Interaction, fruit: str):
    await interaction.response.send_message(f'Your favourite fruit seems t
```

Parameters:

****parameters** – The parameters to mark as autocomplete.

Raises:

`TypeError` – The parameter name is not found or the parameter type was incorrect.

`@discord.app_commands.` **guilds** (** guild_ids*)

Associates the given guilds with the command.

When the command instance is added to a `CommandTree`, the guilds that are specified by this decorator become the default guilds that it's added to rather than being a global command.

> **Note**
>
> Due to an implementation quirk and Python limitation, if this is used in conjunction with the `CommandTree.command()` or `CommandTree.context_menu()` decorator then this must go below that decorator.

v: stable

Example:

```
MY_GUILD_ID = discord.Object(...)  # Guild ID here
```

```
@app_commands.command()
@app_commands.guilds(MY_GUILD_ID)
async def bonk(interaction: discord.Interaction):
    await interaction.response.send_message('Bonk', ephemeral=True)
```

Parameters:

***guild_ids** (Union[`int` , `Snowflake`]) – The guilds to associate this command with. The command tree will use this as the default when added rather than adding it as a global command.

@discord.app_commands. **guild_only** (*func = None*)

A decorator that indicates this command can only be used in a guild context.

This is **not** implemented as a `check()` , and is instead verified by Discord server side. Therefore, there is no error handler called when a command is used within a private message.

This decorator can be called with or without parentheses.

Due to a Discord limitation, this decorator does nothing in subcommands and is ignored.

Examples

```
@app_commands.command()
@app_commands.guild_only()
async def my_guild_only_command(interaction: discord.Interaction) -> None:
    await interaction.response.send_message('I am only available in guilds
```

@discord.app_commands. **default_permissions** (*** perms*)

A decorator that sets the default permissions needed to execute this command.

When this decorator is used, by default users must have these permissions to execute the command. However, an administrator can change the permissions needed to execute this command using the official client. Therefore, this only serves as a hint.

Setting an empty permissions field, including via calling this with no arguments, will disallow anyone except server administrators from using the command in a guild.

This is sent to Discord server side, and is not a `check()` . Therefore, error handlers are not called.

Due to a Discord limitation, this decorator does nothing in subcommands and is ignored.

v: stable

> **Warning**

> This serves as a *hint* and members are *not* required to have the permissions given to actually execute this command. If you want to ensure that members have the permissions needed, consider using `has_permissions()` instead.

Parameters:

****perms** (`bool`) – Keyword arguments denoting the permissions to set as the default.

Example

```
@app_commands.command()
@app_commands.default_permissions(manage_messages=True)
async def test(interaction: discord.Interaction):
    await interaction.response.send_message('You may or may not have manag
```

Checks

@discord.app_commands. **check** (*predicate*)

A decorator that adds a check to an application command.

These checks should be predicates that take in a single parameter taking a `Interaction` . If the check returns a `False` -like value then during invocation a `CheckFailure` exception is raised and sent to the appropriate error handlers.

These checks can be either a coroutine or not.

Examples

Creating a basic check to see if the command invoker is you.

```
def check_if_it_is_me(interaction: discord.Interaction) -> bool:
    return interaction.user.id == 85309593344815104

@tree.command()
@app_commands.check(check_if_it_is_me)
async def only_for_me(interaction: discord.Interaction):
    await interaction.response.send_message('I know you!', ephemeral=True)
```

Transforming common checks into its own decorator:

```
def is_me():
    def predicate(interaction: discord.Interaction) -> bool:
        return interaction.user.id == 85309593344815104
    return app_commands.check(predicate)
```

v: stable

```
@tree.command()
@is_me()
async def only_me(interaction: discord.Interaction):
    await interaction.response.send_message('Only you!')
```

Parameters:

predicate (Callable[[`Interaction`], `bool`]) – The predicate to check if the command should be invoked.

@discord.app_commands.checks.**has_role**(*item*, */*)

A `check()` that is added that checks if the member invoking the command has the role specified via the name or ID specified.

If a string is specified, you must give the exact name of the role, including caps and spelling.

If an integer is specified, you must give the exact snowflake ID of the role.

This check raises one of two special exceptions, `MissingRole` if the user is missing a role, or `NoPrivateMessage` if it is used in a private message. Both inherit from `CheckFailure` .

New in version 2.0.

> **Note**
>
> This is different from the permission system that Discord provides for application commands. This is done entirely locally in the program rather than being handled by Discord.

Parameters:

item (Union[`int` , `str`]) – The name or ID of the role to check.

@discord.app_commands.checks.**has_any_role**(**items*)

A `check()` that is added that checks if the member invoking the command has **any** of the roles specified. This means that if they have one out of the three roles specified, then this check will return `True` .

Similar to `has_role()` , the names or IDs passed in must be exact.

This check raises one of two special exceptions, `MissingAnyRole` if the user is missing all roles, or `NoPrivateMessage` if it is used in a private message. Both inherit from `CheckFailure` .

New in version 2.0.

v: stable

> **Note**
>
> This is different from the permission system that Discord provides for application

This is different from the permission system that Discord provides for application commands. This is done entirely locally in the program rather than being handled by Discord.

Parameters:

items (List[Union[`str` , `int`]]) – An argument list of names or IDs to check that the member has roles wise.

Example

```
@tree.command()
@app_commands.checks.has_any_role('Library Devs', 'Moderators', 492212595
async def cool(interaction: discord.Interaction):
    await interaction.response.send_message('You are cool indeed')
```

@discord.app_commands.checks.**has_permissions**(** *perms*)

A `check()` that is added that checks if the member has all of the permissions necessary.

Note that this check operates on the permissions given by `discord.Interaction.permissions` .

The permissions passed in must be exactly like the properties shown under `discord.Permissions` .

This check raises a special exception, `MissingPermissions` that is inherited from `CheckFailure` .

New in version 2.0.

Note

This is different from the permission system that Discord provides for application commands. This is done entirely locally in the program rather than being handled by Discord.

Parameters:

****perms** (`bool`) – Keyword arguments denoting the permissions to check for.

Example

```
@tree.command()
@app_commands.checks.has_permissions(manage_messages=True)
async def test(interaction: discord.Interaction):
    await interaction.response.send_message('You can manage messages.')
```

v: stable

@discord.app_commands.checks.**bot_has_permissions**(

@discord.app_commands.checks.**bot_has_permissions**(***perms*)

Similar to `has_permissions()` except checks if the bot itself has the permissions listed. This relies on `discord.Interaction.app_permissions`.

This check raises a special exception, `BotMissingPermissions` that is inherited from `CheckFailure`.

New in version 2.0.

@discord.app_commands.checks.**cooldown**(*rate*, *per*, ***, *key=...*)

A decorator that adds a cooldown to a command.

A cooldown allows a command to only be used a specific amount of times in a specific time frame. These cooldowns are based off of the `key` function provided. If a `key` is not provided then it defaults to a user-level cooldown. The `key` function must take a single parameter, the `discord.Interaction` and return a value that is used as a key to the internal cooldown mapping.

The `key` function can optionally be a coroutine.

If a cooldown is triggered, then `CommandOnCooldown` is raised to the error handlers.

Examples

Setting a one per 5 seconds per member cooldown on a command:

```
@tree.command()
@app_commands.checks.cooldown(1, 5.0, key=lambda i: (i.guild_id, i.user.id
async def test(interaction: discord.Interaction):
    await interaction.response.send_message('Hello')

@test.error
async def on_test_error(interaction: discord.Interaction, error: app_comma
    if isinstance(error, app_commands.CommandOnCooldown):
        await interaction.response.send_message(str(error), ephemeral=True
```

Parameters:

- **rate** (`int`) – The number of times a command can be used before triggering a cooldown.
- **per** (`float`) – The amount of seconds to wait for a cooldown when it's been triggered.
- **key** (Optional[Callable[[`discord.Interaction`], `collections.abc.Hashable`]]) – A function that returns a key to the mapping denoting the type of cooldown. Can optionally be a coroutine. If not given then defaults to a user-level cooldown. If `None` is passed then it is interpreted as a "global" cooldown.

@discord.app_commands.checks.**dynamic_cooldown**(*factory*, ***

@discord.app_commands.checks.**dynamic_cooldown**(*factory*, *, *key=...*)

A decorator that adds a dynamic cooldown to a command.

A cooldown allows a command to only be used a specific amount of times in a specific time frame. These cooldowns are based off of the `key` function provided. If a `key` is not provided then it defaults to a user-level cooldown. The `key` function must take a single parameter, the `discord.Interaction` and return a value that is used as a key to the internal cooldown mapping.

If a `factory` function is given, it must be a function that accepts a single parameter of type `discord.Interaction` and must return a `Cooldown` or `None`. If `None` is returned then that cooldown is effectively bypassed.

Both `key` and `factory` can optionally be coroutines.

If a cooldown is triggered, then `CommandOnCooldown` is raised to the error handlers.

Examples

Setting a cooldown for everyone but the owner.

```
def cooldown_for_everyone_but_me(interaction: discord.Interaction) -> Opti
    if interaction.user.id == 80088516616269824:
        return None
    return app_commands.Cooldown(1, 10.0)

@tree.command()
@app_commands.checks.dynamic_cooldown(cooldown_for_everyone_but_me)
async def test(interaction: discord.Interaction):
    await interaction.response.send_message('Hello')

@test.error
async def on_test_error(interaction: discord.Interaction, error: app_comma
    if isinstance(error, app_commands.CommandOnCooldown):
        await interaction.response.send_message(str(error), ephemeral=True
```

Parameters:

- **factory** (Optional[Callable[[`discord.Interaction`], Optional[`Cooldown`]]]) – A function that takes an interaction and returns a cooldown that will apply to that interaction or `None` if the interaction should not have a cooldown.
- **key** (Optional[Callable[[`discord.Interaction`], `collections.abc.Hashable`]]) – A function that returns a key to the mapping denoting the type of cooldown. Can optionally be a coroutine. If not given then defaults to a user-level cooldown. If `None` is passed then it is interpreted as a “global” cooldown.

Cooldown

class discord.app_commands. **Cooldown** (*rate*, *per*)

Attributes

- per
- rate

Methods

- def copy
- def get_retry_after
- def get_tokens
- def reset
- def update_rate_limit

Represents a cooldown for a command.

New in version 2.0.

rate

The total number of tokens available per `per` seconds.

Type:
`float`

per

The length of the cooldown period in seconds.

Type:
`float`

get_tokens (*current* = *None*)

Returns the number of available tokens before rate limiting is applied.

Parameters:
current (Optional[`float`]) – The time in seconds since Unix epoch to calculate tokens at. If not supplied then `time.time()` is used.

Returns:
The number of tokens available before the cooldown is to be applied.

Return type:
`int`

get_retry_after (*current* = *None*)

Returns the time in seconds until the cooldown will be reset.

Parameters:
current (Optional[`float`]) – The current time in seconds since Unix epoch. If not supplied, then `time.time()` is used.

Returns:
The number of seconds to wait before this cooldown will be reset

The number of seconds to wait before this cooldown will be reset.

Return type:

`float`

`update_rate_limit` (*current = None , * , tokens = 1*)

Updates the cooldown rate limit.

Parameters:

- **current** (Optional[`float`]) – The time in seconds since Unix epoch to update the rate limit at. If not supplied, then `time.time()` is used.
- **tokens** (`int`) – The amount of tokens to deduct from the rate limit.

Returns:

The retry-after time in seconds if rate limited.

Return type:

Optional[`float`]

`reset` ( )

Reset the cooldown to its initial state.

`copy` ( )

Creates a copy of this cooldown.

Returns:

A new instance of this cooldown.

Return type:

`Cooldown`

Namespace

class discord.app_commands. **Namespace**

An object that holds the parameters being passed to a command in a mostly raw state.

This class is deliberately simple and just holds the option name and resolved value as a simple key-pair mapping. These attributes can be accessed using dot notation. For example, an option with the name of `example` can be accessed using `ns.example` . If an attribute is not found, then `None` is returned rather than an attribute error.

> **Warning** v: stable
>
> The key names come from the raw Discord data, which means that if a parameter was renamed then the renamed key is used instead of the function parameter name.

New in version 2.0.

Supported Operations

`x == y`

Checks if two namespaces are equal by checking if all attributes are equal.

`x != y`

Checks if two namespaces are not equal.

`x[key]`

Returns an attribute if it is found, otherwise raises a `KeyError`.

`key in x`

Checks if the attribute is in the namespace.

`iter(x)`

Returns an iterator of `(name, value)` pairs. This allows it to be, for example, constructed as a dict or a list of pairs.

This namespace object converts resolved objects into their appropriate form depending on their type. Consult the table below for conversion information.

Option Type	Resolved Type
`AppCommandOptionType.string`	`str`
`AppCommandOptionType.integer`	`int`
`AppCommandOptionType.boolean`	`bool`
`AppCommandOptionType.number`	`float`
`AppCommandOptionType.user`	`User` or `Member`
`AppCommandOptionType.channel`	`AppCommandChannel` or `AppCommandThrea`

`AppCommandOptionType.role`	`Role`
`AppCommandOptionType.mentionable`	`User` or `Member`, or `Role`
`AppCommandOptionType.attachment`	`Attachment`

> **Note**
>
> In autocomplete interactions, the namespace might not be validated or filled in. Discord does not send the resolved data as well, so this means that certain fields end up just as IDs rather than the resolved data. In these cases, a `discord.Object` is returned instead.
>
> This is a Discord limitation.

Transformers

Transformer

class discord.app_commands.**Transformer**

Attributes

- channel_types
- choices
- max_value
- min_value
- type

Methods

- async autocomplete
- async transform

The base class that allows a type annotation in an application command parameter to map into a `AppCommandOptionType` and transform the raw value into one from this type.

This class is customisable through the overriding of methods and properties in the class and by using it as the second type parameter of the `Transform` class. For example, to convert a string into a custom pair type:

```
class Point(typing.NamedTuple):
    x: int
    y: int

class PointTransformer(app_commands.Transformer):
```

v: stable

```
class PointTransformer(app_commands.Transformer):
    async def transform(self, interaction: discord.Interaction, value: str
        (x, _, y) = value.partition(',')
        return Point(x=int(x.strip()), y=int(y.strip()))

@app_commands.command()
async def graph(
    interaction: discord.Interaction,
    point: app_commands.Transform[Point, PointTransformer],
):
    await interaction.response.send_message(str(point))
```

If a class is passed instead of an instance to the second type parameter, then it is constructed with no arguments passed to the `__init__` method.

New in version 2.0.

property **type**

The option type associated with this transformer.

This must be a `property`.

Defaults to `string`.

Type:
`AppCommandOptionType`

property **channel_types**

A list of channel types that are allowed to this parameter.

Only valid if the `type()` returns `channel`.

This must be a `property`.

Defaults to an empty list.

Type:
List[`ChannelType`]

property **min_value**

The minimum supported value for this parameter.

Only valid if the `type()` returns `number` `integer`, or `string`.

This must be a `property`.

Defaults to `None`.

v: stable

Type:
Optional[`int`]

property **max_value**

The maximum supported value for this parameter.

Only valid if the `type()` returns `number` `integer`, or `string`.

This must be a `property`.

Defaults to `None`.

Type:
Optional[`int`]

property **choices**

A list of up to 25 choices that are allowed to this parameter.

Only valid if the `type()` returns `number` `integer`, or `string`.

This must be a `property`.

Defaults to `None`.

Type:
Optional[List[`Choice`]]

await **transform**(*interaction*, *value*, */*)

This function *could be a coroutine*.

Transforms the converted option value into another value.

The value passed into this transform function is the same as the one in the `conversion table`.

Parameters:

- **interaction** (`Interaction`) – The interaction being handled.
- **value** (*Any*) – The value of the given argument after being resolved. See the `conversion table` for how certain option types correspond to certain values.

await **autocomplete**(*interaction*, *value*, */*)

This function is a *coroutine*.

An autocomplete prompt handler to be automatically used by options using this transformer.

> **Note**
>
> Autocomplete is only supported for options with a `type()` of `string`, `integer`, or `number`.

Parameters:

- **interaction** (`Interaction`) – The autocomplete interaction being handled.
- **value** (Union[`str` , `int` , `float`]) – The current value entered by the user.

Returns:

A list of choices to be displayed to the user, a maximum of 25.

Return type:

List[`Choice`]

Transform

class discord.app_commands. **Transform**

A type annotation that can be applied to a parameter to customise the behaviour of an option type by transforming with the given `Transformer` . This requires the usage of two generic parameters, the first one is the type you're converting to and the second one is the type of the `Transformer` actually doing the transformation.

During type checking time this is equivalent to `typing.Annotated` so type checkers understand the intent of the code.

For example usage, check `Transformer` .

New in version 2.0.

Range

class discord.app_commands. **Range**

A type annotation that can be applied to a parameter to require a numeric or string type to fit within the range provided.

During type checking time this is equivalent to `typing.Annotated` so type checkers understand the intent of the code.

Some example ranges:

- `Range[int, 10]` means the minimum is 10 with no maximum.
- `Range[int, None, 10]` means the maximum is 10 with no minimum.
- `Range[int, 1, 10]` means the minimum is 1 and the maximum is 10.
- `Range[float, 1.0, 5.0]` means the minimum is 1.0 and the maximum is 5.0.
- `Range[str, 1, 10]` means the minimum length is 1 and the maximum length is 10.

New in version 2.0.

Examples

```
@app_commands.command()
async def range(interaction: discord.Interaction, value: app_commands.Rang
    await interaction.response.send_message(f'Your value is {value}', ephe
```

Translations

Translator

class discord.app_commands.**Translator**

Methods

async load
async translate
async unload

A class that handles translations for commands, parameters, and choices.

Translations are done lazily in order to allow for async enabled translations as well as supporting a wide array of translation systems such as `gettext` and Project Fluent.

In order for a translator to be used, it must be set using the `CommandTree.set_translator()` method. The translation flow for a string is as follows:

1. Use `locale_str` instead of `str` in areas of a command you want to be translated.
    - Currently, these are command names, command descriptions, parameter names, parameter descriptions, and choice names.
    - This can also be used inside the `describe()` decorator.
2. Call `CommandTree.set_translator()` to the translator instance that will handle the translations.
3. Call `CommandTree.sync()`
4. The library will call `Translator.translate()` on all the relevant strings being translated.

New in version 2.0.

await **load**()

This function is a *coroutine*.

An asynchronous setup function for loading the translation system.

The default implementation does nothing.

This is invoked when `CommandTree.set_translator()` is called.

await **unload**()

This function is a *coroutine*.

An asynchronous teardown function for unloading the translation system.

The default implementation does nothing.

This is invoked when `CommandTree.set_translator()` is called if a tree already has a translator or when `discord.Client.close()` is called.

await **translate**(*string*, *locale*, *context*)

This function is a *coroutine*.

Translates the given string to the specified locale.

If the string cannot be translated, `None` should be returned.

The default implementation returns `None`.

If an exception is raised in this method, it should inherit from `TranslationError`. If it doesn't, then when this is called the exception will be chained with it instead.

Parameters:

- **string** (`locale_str`) – The string being translated.
- **locale** (`Locale`) – The locale being requested for translation.
- **context** (`TranslationContext`) – The translation context where the string originated from. For better type checking ergonomics, the `TranslationContextTypes` type can be used instead to aid with type narrowing. It is functionally equivalent to `TranslationContext`.

locale_str

class discord.app_commands.**locale_str**(*message*, */*, ** *kwargs*)

Attributes

extras
message

v: stable

Marks a string as ready for translation.

This is done lazily and is not actually translated until `CommandTree.sync()` is called.

The sync method then ultimately defers the responsibility of translating to the `Translator` instance used by the `CommandTree`. For more information on the translation flow, see the `Translator` documentation.

Supported Operations

`str(x)`

Returns the message passed to the string.

`x == y`

Checks if the string is equal to another string.

`x != y`

Checks if the string is not equal to another string.

`hash(x)`

Returns the hash of the string.

New in version 2.0.

`message`

The message being translated. Once set, this cannot be changed.

> **⚠ Warning**
>
> This must be the default "message" that you send to Discord. Discord sends this message back to the library and the library uses it to access the data in order to dispatch commands.
>
> For example, in a command name context, if the command name is `foo` then the message *must* also be `foo`. For other translation systems that require a message ID such as Fluent, consider using a keyword argument to pass it in.

Type:
`str`

`extras`

A dict of user provided extras to attach to the translated string. This can be used to add more context, information, or any metadata necessary to aid in actually translating the string.

Since these are passed via keyword arguments, the keys are strings.

Type:

Type:
dict

TranslationContext

class discord.app_commands.**TranslationContext**(*location*, *data*)

Attributes

data
location

A class that provides context for the `locale_str` being translated.

This is useful to determine where exactly the string is located and aid in looking up the actual translation.

location

The location where this string is located.

Type:
`TranslationContextLocation`

data

The extraneous data that is being translated.

Type:
Any

TranslationContextLocation

class discord.app_commands.**TranslationContextLocation**

An enum representing the location context that the translation occurs in when requested for translation.

New in version 2.0.

command_name

The translation involved a command name.

command_description

The translation involved a command description.

group_name

The translation involved a group name

The translation involved a group name.

group_description

The translation involved a group description.

parameter_name

The translation involved a parameter name.

parameter_description

The translation involved a parameter description.

choice_name

The translation involved a choice name.

other

The translation involved something else entirely. This is useful for running `Translator.translate()` for custom usage.

Exceptions

exception discord.app_commands.**AppCommandError**

The base exception type for all application command related errors.

This inherits from `discord.DiscordException`.

This exception and exceptions inherited from it are handled in a special way as they are caught and passed into various error handlers in this order:

- `Command.error`
- `Group.on_error`
- `CommandTree.on_error`

New in version 2.0.

exception discord.app_commands.**CommandInvokeError**(*command*, *e*)

An exception raised when the command being invoked raised an exception.

This inherits from `AppCommandError`.

New in version 2.0.

original v: stable

The original exception that was raised. You can also get this via the `__cause__` attribute.

Type:
Exception

command

The command that failed.

Type:
Union[Command , ContextMenu]

exception discord.app_commands. **TransformerError** (*value* , *opt_type* , *transformer*)

An exception raised when a Transformer or type annotation fails to convert to its target type.

This inherits from AppCommandError .

If an exception occurs while converting that does not subclass AppCommandError then the exception is wrapped into this exception. The original exception can be retrieved using the __cause__ attribute. Otherwise if the exception derives from AppCommandError then it will be propagated as-is.

New in version 2.0.

value

The value that failed to convert.

Type:
Any

type

The type of argument that failed to convert.

Type:
AppCommandOptionType

transformer

The transformer that failed the conversion.

Type:
Transformer

exception discord.app_commands. **TranslationError** (* *msg* , *string* = *None* , *locale* = *None* , *context*)

An exception raised when the library fails to translate a string.

This inherits from AppCommandError .

If an exception occurs while calling Translator.translate() that does not subclass this

then the exception is wrapped into this exception. The original exception can be retrieved using the `__cause__` attribute. Otherwise it will be propagated as-is.

New in version 2.0.

string

The string that caused the error, if any.

Type:
Optional[Union[`str` , `locale_str`]]

locale

The locale that caused the error, if any.

Type:
Optional[`Locale`]

context

The context of the translation that triggered the error.

Type:
`TranslationContext`

exception discord.app_commands. **CheckFailure**

An exception raised when check predicates in a command have failed.

This inherits from `AppCommandError` .

New in version 2.0.

exception discord.app_commands. **NoPrivateMessage** (*message* = *None*)

An exception raised when a command does not work in a direct message.

This inherits from `CheckFailure` .

New in version 2.0.

exception discord.app_commands. **MissingRole** (*missing_role*)

An exception raised when the command invoker lacks a role to run a command.

This inherits from `CheckFailure` .

New in version 2.0.

missing_role

The required role that is missing. This is the parameter passed to `has_ro` v: stable

Type:
Union[`str` , `int`]

exception discord.app_commands.**MissingAnyRole**(*missing_roles*)

An exception raised when the command invoker lacks any of the roles specified to run a command.

This inherits from `CheckFailure`.

New in version 2.0.

missing_roles

The roles that the invoker is missing. These are the parameters passed to `has_any_role()`.

Type:
List[Union[`str`, `int`]]

exception discord.app_commands.**MissingPermissions**(*missing_permissions*, **args*)

An exception raised when the command invoker lacks permissions to run a command.

This inherits from `CheckFailure`.

New in version 2.0.

missing_permissions

The required permissions that are missing.

Type:
List[`str`]

exception discord.app_commands.**BotMissingPermissions**(*missing_permissions*, **args*)

An exception raised when the bot's member lacks permissions to run a command.

This inherits from `CheckFailure`.

New in version 2.0.

missing_permissions

The required permissions that are missing.

Type:
List[`str`]

exception discord.app_commands.**CommandOnCooldown**(*cooldo* *retry_after*)

v: stable

An exception raised when the command being invoked is on cooldown

An exception raised when the command being invoked is on cooldown.

This inherits from `CheckFailure`.

New in version 2.0.

cooldown

The cooldown that was triggered.

Type:
`Cooldown`

retry_after

The amount of seconds to wait before you can retry again.

Type:
`float`

exception discord.app_commands.**CommandLimitReached**(*guild_id*, *limit*, *type=<AppCommandType.chat_input: 1>*)

An exception raised when the maximum number of application commands was reached either globally or in a guild.

This inherits from `AppCommandError`.

New in version 2.0.

type

The type of command that reached the limit.

Type:
`AppCommandType`

guild_id

The guild ID that reached the limit or `None` if it was global.

Type:
Optional[`int`]

limit

The limit that was hit.

Type:
`int`

exception discord.app_commands.**CommandAlreadyRegistered**(*name*, *guild_id*)

An exception raised when a command is already registered.

This inherits from AppCommandError

This inherits from AppCommandError .

New in version 2.0.

name

The name of the command already registered.

Type:
str

guild_id

The guild ID this command was already registered at. If None then it was a global command.

Type:
Optional[int]

exception discord.app_commands. **CommandSignatureMismatch** (*command*)

An exception raised when an application command from Discord has a different signature from the one provided in the code. This happens because your command definition differs from the command definition you provided Discord. Either your code is out of date or the data from Discord is out of sync.

This inherits from AppCommandError .

New in version 2.0.

command

The command that had the signature mismatch.

Type:
Union[Command , ContextMenu , Group]

exception discord.app_commands. **CommandNotFound** (*name* , *parents* , *type=<AppCommandType.chat_input: 1>*)

An exception raised when an application command could not be found.

This inherits from AppCommandError .

New in version 2.0.

name

The name of the application command not found.

Type:
str

v: stable

parents

A list of parent command names that were previously found prior to the application command not being found.

Type:
List[`str`]

type

The type of command that was not found.

Type:
`AppCommandType`

exception discord.app_commands. **MissingApplicationID** (*message* = *None*)

An exception raised when the client does not have an application ID set. An application ID is required for syncing application commands.

This inherits from `AppCommandError` .

New in version 2.0.

exception discord.app_commands. **CommandSyncFailure** (*child* , *commands*)

An exception raised when `CommandTree.sync()` failed.

This provides syncing failures in a slightly more readable format.

This inherits from `AppCommandError` and `HTTPException` .

New in version 2.0.

Exception Hierarchy

- » `DiscordException`
  - » `AppCommandError`
    - » `CommandInvokeError`
    - » `TransformerError`
    - » `TranslationError`
    - » `CheckFailure`
      - » `NoPrivateMessage`
      - » `MissingRole`
      - » `MissingAnyRole`
      - » `MissingPermissions`
      - » `BotMissingPermissions`
      - » `CommandOnCooldown`
      - `CommandLimitReached`

- » CommandLimitReached
- » CommandAlreadyRegistered
- » CommandSignatureMismatch
- » CommandNotFound
- » MissingApplicationID
- » CommandSyncFailure

» HTTPException

- » CommandSyncFailure