
Function Cooldowns

Skelmis

Aug 08, 2023

MAIN:

1	Decorator Usage	3
2	Utility Usage	5
3	Buckets	9
4	Library Exceptions	11
5	Example Usage	13
5.1	Example bot usage	13
5.2	Example cog usage	13
5.3	Handling cooldown's	14
5.4	Get Remaining Calls	14
5.5	Reset a specific cooldown	14
5.6	Reset all cooldowns on a callable	15
5.7	Reset a specific bucket on a cooldown	15
5.8	Cooldown checks	15
5.9	Custom buckets	16
5.10	Example async custom buckets	16
5.11	Stacking cooldown's	16
5.12	Shared cooldowns	17
5.13	Manually using cooldowns	17
6	Nextcord Slash Cooldown's	19
6.1	Decorator arguments	19
6.2	Possible buckets	20
6.3	Example usage	21
6.4	Example cog usage	21
6.5	Handling cooldown's	21
6.6	Get remaining calls	22
6.7	Cooldown checks	23
6.8	Custom buckets	23
6.9	Stacking cooldown's	24
7	Extra things	25
8	Cooldown Reference	27
9	TimesPer Reference	31
10	Protocol Reference	33

11 HashableArguments Reference	35
12 Indices and tables	37
Index	39

Cooldowns for Coroutines, simple as that.

DECORATOR USAGE

You have two choices for decorators, as documented below.

For example usages, view the examples section.

```
cooldowns.cooldown(limit: int, time_period: float | ~datetime.timedelta, bucket:
    ~cooldowns.protocols.bucket.CooldownBucketProtocol |
    ~cooldowns.protocols.bucket.AsyncCooldownBucketProtocol, check:
    ~typing.Callable[[~typing.Any, ~typing.Any], ~typing.Coroutine[~typing.Any, ~typing.Any,
    ~typing.Any]] | None = <function default_check>, *, cooldown_id: int | str | None = None)
```

Wrap this Callable in a cooldown.

Parameters

- **limit** (*int*) – How many call's can be made in the time period specified by `time_period`
- **time_period** (*Union[[float](#), [datetime.timedelta](#)]*) – The time period related to `limit`. This is seconds.
- **bucket** (*Union[[CooldownBucketProtocol](#), [AsyncCooldownBucketProtocol](#)]*) – The `Bucket` implementation to use as a bucket to separate cooldown buckets.
- **check** (*Optional[[MaybeCoro](#)]*) – A Callable which dictates whether to apply the cooldown on current invoke.

If this Callable returns a truthy value, then the cooldown will be used for the current call.

I.e. If you wished to bypass cooldowns, you would return `False` if you invoked the Callable.

Note: This check will be given the same arguments as the item you are applying the cooldown to.

- **cooldown_id** (*Optional[Union[[int](#), [str](#)]]*) – Useful for resetting individual stacked cooldowns. This should be unique globally, behaviour is not guaranteed if not unique.

Raises

- **[RuntimeError](#)** – Expected the decorated function to be a coroutine
- **[CallableOnCooldown](#)** – This call resulted in a cooldown being put into effect

```
cooldowns.static_cooldown(limit: int, reset_times: ~datetime.time | ~typing.List[~datetime.time], bucket:
    ~cooldowns.protocols.bucket.CooldownBucketProtocol |
    ~cooldowns.protocols.bucket.AsyncCooldownBucketProtocol, check:
    ~typing.Callable[[~typing.Any, ~typing.Any], ~typing.Coroutine[~typing.Any,
    ~typing.Any, ~typing.Any]] | None = <function default_check>, *, cooldown_id:
    int | str | None = None)
```

Parameters

- **limit** (*int*) – How many call's can be made in the time period specified by `time_period`
- **reset_times** (*Union[datetime.time, List[datetime.time]]*) – A time or list of the possible times in the day to reset cooldowns at
- **bucket** (*Optional[CooldownBucketProtocol]*) – The Bucket implementation to use as a bucket to separate cooldown buckets.

Defaults to `CooldownBucket.all`

- **func** (*Optional[Callable]*) – The function this cooldown is attached to
- **cooldown_id** (*Optional[Union[int, str]]*) – Useful for resetting individual stacked cooldowns. This should be unique globally, behaviour is not guaranteed if not unique.
- **check** (*Optional[MaybeCoro]*) – The check used to validate calls to this Cooldown.

This is not used here, however, its required as an implementation detail for shared cooldowns and can be safely ignored as a parameter.

Note: This check will be given the same arguments as the item you are applying the cooldown to.

Notes

All times are internally handled based off UTC.

Raises

- **RuntimeError** – Expected the decorated function to be a coroutine
- **CallableOnCooldown** – This call resulted in a cooldown being put into effect

`cooldowns.shared_cooldown(cooldown_id: int | str | None)`

Wrap this Callable in a shared cooldown.

Use `define_shared_cooldown()` before this.

Parameters

cooldown_id (*Optional[Union[int, str]]*) – The cooldown for the registered shared cooldown.

Raises

- **RuntimeError** – Expected the decorated function to be a coroutine
- **CallableOnCooldown** – This call resulted in a cooldown being put into effect
- **NonExistent** – Could not find a cooldown with this ID registered.

UTILITY USAGE

Various utilities for using cooldowns.

```
cooldowns.define_shared_cooldown(limit: int, time_period: float, bucket: Union[CooldownBucketProtocol, AsyncCooldownBucketProtocol], cooldown_id: COOLDOWN_ID, *, check: Optional[MaybeCoro] = <function default_check>)
```

Define a global cooldown which can be used to ratelimit 1 or more callables under the same situations.

View the examples for how to use this.

Parameters

- **limit** (*int*) – How many call's can be made in the time period specified by **time_period**
- **time_period** (*float*) – The time period related to **limit**
- **bucket** (*Union[CooldownBucketProtocol, AsyncCooldownBucketProtocol]*) – The **Bucket** implementation to use as a bucket to separate cooldown buckets.
- **cooldown_id** (*Union[int, str]*) – The ID used to refer to this when defining a **shared_cooldown**

This should be unique globally, behaviour is not guaranteed if not unique.

- **check** (*Optional[MaybeCoro]*) – A Callable which dictates whether or not to apply the cooldown on current invoke.

If this Callable returns a truthy value, then the cooldown will be used for the current call.

I.e. If you wished to bypass cooldowns, you would return `False` if you invoked the Callable.

Raises

CooldownAlreadyExists – A Cooldown with this ID already exists.

Notes

All times are internally handled based off UTC.

```
cooldowns.define_shared_static_cooldown(limit: int, reset_times: Union[datetime.time, List[datetime.time]], bucket: Union[CooldownBucketProtocol, AsyncCooldownBucketProtocol], cooldown_id: COOLDOWN_ID, *, check: Optional[MaybeCoro] = <function default_check>)
```

Define a global cooldown which can be used to ratelimit 1 or more callables under the same situations.

View the examples for how to use this.

Parameters

- **limit** (*int*) – How many call's can be made in the time period specified by *time_period*
- **reset_times** (*Union[datetime.time, List[datetime.time]]*) – A time or list of the possible times in the day to reset cooldowns at
- **bucket** (*Union[CooldownBucketProtocol, AsyncCooldownBucketProtocol]*) – The Bucket implementation to use as a bucket to separate cooldown buckets.
- **cooldown_id** (*Union[int, str]*) – The ID used to refer to this when defining a shared_cooldown

This should be unique globally, behaviour is not guaranteed if not unique.

- **check** (*Optional[MaybeCoro]*) – A Callable which dictates whether or not to apply the cooldown on current invoke.

If this Callable returns a truthy value, then the cooldown will be used for the current call.

I.e. If you wished to bypass cooldowns, you would return False if you invoked the Callable.

Raises

CooldownAlreadyExists – A Cooldown with this ID already exists.

async `cooldowns.get_remaining_calls(func: Callable[[Any, Any], Coroutine[Any, Any, Any]], *args, **kwargs) → int`

Given a *Callable*, return the amount of remaining available calls before these arguments will result in the callable being rate-limited.

Parameters

- **func** (*MaybeCoro*) – The *Callable* you want to check.
- **args** – Any arguments you will pass.
- **kwargs** – Any key-word arguments you will pass.

Returns

How many more times this *Callable* can be called without being rate-limited.

Return type

int

Raises

NoRegisteredCooldowns – The given *Callable* has no cooldowns.

Notes

This aggregates all attached cooldowns and returns the lowest remaining amount.

`cooldowns.reset_cooldowns(func: Callable[[Any, Any], Coroutine[Any, Any, Any]])`

Reset all cooldown's on this *Callable* back to default settings.

Parameters

func (*MaybeCoro*) – The func with cooldowns we should reset.

Raises

NoRegisteredCooldowns – The func has no cooldown's attached.

`cooldowns.reset_cooldown(cooldown_id: int | str)`

Reset the cooldown denoted by *cooldown_id*.

Parameters

cooldown_id (*Union[int, str]*) – The id of the cooldown we wish to reset

Raises

NonExistent – Cannot find a cooldown with this id.

async `cooldowns.reset_bucket(func: Callable[[Any, Any], Coroutine[Any, Any]], *args, **kwargs)`

Reset all buckets matching the provided arguments.

Parameters

- **func** (*MaybeCoro*) – The func with cooldowns we should reset.
- **args** – Any arguments you will pass.
- **kwargs** – Any key-word arguments you will pass.

Notes

Does nothing if it resets nothing.

`cooldowns.get_cooldown(func: MaybeCoro, cooldown_id: COOLDOWN_ID) → CooldownT`

Get the *Cooldown* object from the func with the provided cooldown id.

Parameters

- **func** (*MaybeCoro*) – The func with this cooldown.
- **cooldown_id** (*Union[int, str]*) – The id of the cooldown we wish to get

Returns

The associated cooldown

Return type

Cooldown

Raises

NonExistent – Failed to find that cooldown on this func.

`cooldowns.get_shared_cooldown(cooldown_id: COOLDOWN_ID) → CooldownT`

Retrieve a shared *Cooldown* or *StaticCooldown* object.

Parameters

cooldown_id (*Union[int, str]*) – The id of the cooldown we wish to get

Returns

The associated cooldown

Return type

Cooldown

Raises

NonExistent – Failed to find that cooldown

`cooldowns.get_all_cooldowns(func: MaybeCoro) → List[CooldownT]`

Get all the *Cooldown* objects from the func provided.

Parameters

func (*MaybeCoro*) – The func with this cooldown.

Returns

The associated cooldowns

Return type

List[*Cooldown*]

Raises

NoRegisteredCooldowns – No cooldowns on this func

BUCKETS

The current bucket offerings which are built in.

Note, this package is not dependant on Nextcord and if you do not have it there won't be any issues if you don't use it.

class `cooldowns.buckets.CooldownBucket`(*value*)

A collection of generic CooldownBucket's for usage in cooldown's.

See `cooldowns.protocols.bucket.CooldownBucketProtocol`

all

The buckets are defined using all arguments passed to the *Callable*

args

The buckets are defined using all non-keyword arguments passed to the *Callable*

kwargs

The buckets are defined using all keyword arguments passed to the *Callable*

This requires a full import, and Nextcord.

class `cooldowns.buckets.slash.SlashBucket`(*value*)

A collection of generic cooldown bucket's for usage with nextcord slash commands which take *Interaction*

author

Rate-limits the command per person.

guild

Rate-limits the command per guild.

channel

Rate-limits the command per channel

command

Rate-limits the entire command as one.

LIBRARY EXCEPTIONS

All exceptions inherit from `BaseCooldownException`

class `cooldowns.exceptions.BaseCooldownException(*args)`

A base exception handler.

class `cooldowns.exceptions.NonExistent(*args)`

There doesn't already exist a bucket for this.

class `cooldowns.exceptions.CooldownAlreadyExists(*args)`

A Cooldown with this ID already exists.

class `cooldowns.exceptions.UnknownBucket(*args)`

Failed to process the bucket.

class `cooldowns.exceptions.NoRegisteredCooldowns`

This *Callable* has no attached cooldown's.

class `cooldowns.exceptions.CallableOnCooldown(func: Callable, cooldown: Cooldown, resets_at: datetime.datetime)`

This *Callable* is currently on cooldown.

func

The *Callable* which is currently rate-limited

Type

Callable

cooldown

The Cooldown which applies to the current cooldown

Type

Cooldown

resets_at

The exact datetime this cooldown resets.

Type

datetime.datetime

property retry_after: float

How many seconds before you can retry the *Callable*

Returns

How many seconds before you can retry this

Note: This will be 0 if you can retry now

Return type

float

EXAMPLE USAGE

These can be used on any function or method marked with `async`, Nextcord is just used for the examples here.

5.1 Example bot usage

Rate-limits the command to 1 call per person every 15 seconds in your main file.

```
1 import cooldowns
2
3 ...
4
5 @bot.slash_command(
6     description="Ping command",
7 )
8 @cooldowns.cooldown(1, 15, bucket=cooldowns.SlashBucket.author)
9 async def ping(interaction: nextcord.Interaction):
10     await interaction.response.send_message("Pong!")
```

5.2 Example cog usage

Rate-limits the command to 1 call per guild every 30 seconds.

```
1 import cooldowns
2
3 ...
4
5 @nextcord.slash_command(
6     description="Ping command",
7 )
8 @cooldowns.cooldown(1, 30, bucket=cooldowns.SlashBucket.guild)
9 async def ping(self, interaction: nextcord.Interaction):
10     await interaction.response.send_message("Pong!")
```

5.3 Handling cooldown's

Here is an example error handler

```
1 from cooldowns import CallableOnCooldown
2
3 ...
4
5 @bot.event
6 async def on_application_command_error(inter: nextcord.Interaction, error):
7     error = getattr(error, "original", error)
8
9     if isinstance(error, CallableOnCooldown):
10         await inter.send(
11             f"You are being rate-limited! Retry in `{error.retry_after}` seconds."
12         )
13
14     else:
15         raise error
```

5.4 Get Remaining Calls

```
1 from cooldowns import get_remaining_calls, cooldown, SlashBucket
2
3 @bot.slash_command()
4 @cooldown(2, 10, SlashBucket.command)
5 async def test(inter):
6     ...
7     calls_left = get_remaining_calls(test, inter)
8     await inter.send(f"You can call this {calls_left} times before getting rate-limited")
```

5.5 Reset a specific cooldown

Only resets cooldowns for the given id.

```
1 from cooldowns import cooldown, CooldownBucket, reset_cooldown
2
3 @cooldown(1, 30, CooldownBucket.all, cooldown_id="my_cooldown")
4 async def test(*args, **kwargs):
5     ...
6
7 # Reset
8 reset_cooldown("my_cooldown")
```

5.6 Reset all cooldowns on a callable

Resets all cooldowns on the provided callable.

```

1 from cooldowns import cooldown, CooldownBucket, reset_cooldowns
2
3 @cooldown(1, 30, CooldownBucket.all)
4 @cooldown(1, 15, CooldownBucket.args)
5 async def test(*args, **kwargs):
6     ...
7
8 # Reset
9 reset_cooldowns(test)

```

5.7 Reset a specific bucket on a cooldown

Resets only the given buckets on a cooldown.

```

1 from cooldowns import cooldown, CooldownBucket, reset_bucket
2
3 @cooldown(1, 30, CooldownBucket.all)
4 async def test(*args, **kwargs):
5     ...
6
7     ...
8
9 # Reset the bucket with `1` as an argument
10 reset_bucket(test, 1)

```

5.8 Cooldown checks

Here's an example check to only apply a cooldown if the first argument is equal to 1.

```

1 @cooldown(
2     1, 1, bucket=CooldownBucket.args, check=lambda *args, **kwargs: args[0] == 1
3 )
4 async def test_func(*args, **kwargs) -> (tuple, dict):
5     return args, kwargs

```

Here's one use an async check. Functionally its the same as the previous one.

```

1 async def mock_db_check(*args, **kwargs):
2     # You can do database calls here or anything
3     # since this is an async context
4     return args[0] == 1
5
6 @cooldown(1, 1, bucket=CooldownBucket.args, check=mock_db_check)
7 async def test_func(*args, **kwargs) -> (tuple, dict):
8     return args, kwargs

```

5.9 Custom buckets

All you need is an enum with the process method.

Heres an example which rate-limits based off of the first argument.

```
1 class CustomBucket(Enum):
2     first_arg = 1
3
4     def process(self, *args, **kwargs):
5         if self is CustomBucket.first_arg:
6             # This bucket is based ONLY off
7             # of the first argument passed
8             return args[0]
9
10 # Then to use
11 @cooldown(1, 1, bucket=CustomBucket.first_arg)
12 async def test_func(*args, **kwargs):
13     ....
```

5.10 Example async custom buckets

How to use async buckets to process data

All you need is an enum with the process method which is async.

Here is an example which rate-limits based off of the first argument.

```
1 class CustomBucket(Enum):
2     first_arg = 1
3
4     async def process(self, *args, **kwargs):
5         if self is CustomBucket.first_arg:
6             # This bucket is based ONLY off
7             # of the first argument passed
8             return args[0]
9
10 # Then to use
11 @cooldown(1, 1, bucket=CustomBucket.first_arg)
12 async def test_func(*args, **kwargs):
13     ....
```

5.11 Stacking cooldown's

Stack as many cooldown's as you want, just note Python starts from the bottom decor and works its way up.

```
1 # Can call ONCE time_period second using the same args
2 # Can call TWICE time_period second using the same kwargs
3 @cooldown(1, 1, bucket=CooldownBucket.args)
4 @cooldown(2, 1, bucket=CooldownBucket.kwargs)
```

(continues on next page)

(continued from previous page)

```
5 async def test_func(*args, **kwargs) -> (tuple, dict):  
6     return args, kwargs
```

5.12 Shared cooldowns

This allows you to use the same cooldown on multiple callables.

```
1 from cooldowns import define_shared_cooldown, shared_cooldown, CooldownBucket  
2  
3 define_shared_cooldown(1, 5, CooldownBucket.all, cooldown_id="my_id")  
4  
5 @shared_cooldown("my_id")  
6 async def test_1(*args, **kwargs):  
7     return 1  
8  
9 @shared_cooldown("my_id")  
10 async def test_2(*args, **kwargs):  
11     return 2  
12  
13 # These now both share the same cooldown
```

5.13 Manually using cooldowns

How to use the Cooldown object without a decorator.

```
1 from cooldowns import Cooldown, CooldownBucket  
2  
3 cooldown = Cooldown(1, 5, CooldownBucket.args)  
4  
5 async with cooldown(*args, **kwargs):  
6     # This will apply the cooldown  
7     ...  
8     # Do things
```


NEXTCORD SLASH COOLDOWN'S

Given the lack of official support, here is how to use cooldown's (unofficially).

```
pip install function-cooldowns
```

Check out the Github [here](#)

Want support? Broke something? Ask me in the nextcord discord, Skelmis#9135

6.1 Decorator arguments

```
1 def cooldown(  
2     limit: int,  
3     time_period: float,  
4     bucket: CooldownBucketProtocol,  
5     check: Optional[MaybeCoro] = lambda *args, **kwargs: True,  
6 ):
```

limit: int

How many call's can be made in the time period specified by `time_period`

time_period: float

The time period related to `limit`

bucket: CooldownBucketProtocol

The Bucket implementation to use as a bucket to separate cooldown buckets.

check: Optional[MaybeCoro]

A Callable which dictates whether or not to apply the cooldown on current invoke.

If this Callable returns a truthy value, then the cooldown will be used for the current call.

I.e. If you wished to bypass cooldowns, you would return False if you invoked the Callable.

6.2 Possible buckets

Most basic bucket.

```
1 from cooldowns import CooldownBucket
2
3 CooldownBucket.all
4 CooldownBucket.args
5 CooldownBucket.kwargs
6
7 """
8 A collection of generic CooldownBucket's for usage in cooldown's.
9
10 Attributes
11 =====
12 all
13     The buckets are defined using all
14     arguments passed to the :type:`Callable`
15 args
16     The buckets are defined using all
17     non-keyword arguments passed to the :type:`Callable`
18 kwargs
19     The buckets are defined using all
20     keyword arguments passed to the :type:`Callable`
21 """
```

Slash command bucket.

```
1 from cooldowns import SlashBucket
2
3 SlashBucket.author
4 SlashBucket.guild
5 SlashBucket.channel
6 SlashBucket.command
7
8 """
9 A collection of generic cooldown bucket's for usage
10 with nextcord slash commands which take ``Interaction``
11
12 Attributes
13 =====
14 author
15     Rate-limits the command per person.
16 guild
17     Rate-limits the command per guild.
18 channel
19     Rate-limits the command per channel
20 command
21     Rate-limits the entire command as one.
22 """
```


6.3 Example usage

Rate-limits the command to 1 call per person every 15 seconds in your main file.

```

1 import cooldowns
2
3 ...
4
5 @bot.slash_command(
6     description="Ping command",
7 )
8 @cooldowns.cooldown(1, 15, bucket=cooldowns.SlashBucket.author)
9 async def ping(interaction: nextcord.Interaction):
10     await interaction.response.send_message("Pong!")

```

6.4 Example cog usage

Rate-limits the command to 1 call per guild every 30 seconds.

```

1 import cooldowns
2
3 ...
4
5 @nextcord.slash_command(
6     description="Ping command",
7 )
8 @cooldowns.cooldown(1, 30, bucket=cooldowns.SlashBucket.guild)
9 async def ping(self, interaction: nextcord.Interaction):
10     await interaction.response.send_message("Pong!")

```

6.5 Handling cooldown's

Here is an example error handler

```

1 from cooldowns import CallableOnCooldown
2
3 ...
4
5 @bot.event
6 async def on_application_command_error(inter: nextcord.Interaction, error):
7     error = getattr(error, "original", error)
8
9     if isinstance(error, CallableOnCooldown):
10         await inter.send(
11             f"You are being rate-limited! Retry in `{error.retry_after}` seconds."
12         )
13
14     else:
15         raise error

```

The error `CallableOnCooldown` has the following attributes.

func: Callable

The *Callable* which is currently rate-limited

cooldown: Cooldown

The *Cooldown* which applies to the current cooldown

retry_after: float

How many seconds before you can retry the *Callable*

resets_at: datetime.datetime

The exact datetime this cooldown resets.

6.6 Get remaining calls

Definition

```
1 def get_remaining_calls(func: MaybeCoro, *args, **kwargs) -> int:
2     """
3     Given a :type:`Callable`, return the amount of remaining
4     available calls before these arguments will result
5     in the callable being rate-limited.
6
7     Parameters
8     -----
9     func: MaybeCoro
10         The :type:`Callable` you want to check.
11     args
12         Any arguments you will pass.
13     kwargs
14         Any key-word arguments you will pass.
15
16     Returns
17     -----
18     int
19         How many more times this :type:`Callable`
20         can be called without being rate-limited.
21
22     Raises
23     -----
24     NoRegisteredCooldowns
25         The given :type:`Callable` has no cooldowns.
26
27     Notes
28     -----
29     This aggregates all attached cooldowns
30     and returns the lowest remaining amount.
31     """
```

Example usage

```
1 from cooldowns import get_remaining_calls, cooldown, SlashBucket
2
```

(continues on next page)

(continued from previous page)

```

3 @bot.slash_command()
4 @cooldown(2, 10, SlashBucket.command)
5 async def test(inter):
6     ...
7     calls_left = get_remaining_calls(test, inter)
8     await inter.send(f"You can call this {calls_left} times before getting rate-limited")

```

6.7 Cooldown checks

Here's an example check to only apply a cooldown if the first argument is equal to 1.

```

1 @cooldown(
2     1, 1, bucket=CooldownBucket.args, check=lambda *args, **kwargs: args[0] == 1
3 )
4 async def test_func(*args, **kwargs) -> (tuple, dict):
5     return args, kwargs

```

Here's one use an async check. Functionally its the same as the previous one.

```

1 async def mock_db_check(*args, **kwargs):
2     # You can do database calls here or anything
3     # since this is an async context
4     return args[0] == 1
5
6 @cooldown(1, 1, bucket=CooldownBucket.args, check=mock_db_check)
7 async def test_func(*args, **kwargs) -> (tuple, dict):
8     return args, kwargs

```

6.8 Custom buckets

All you need is an enum with the process method.

Here's an example which rate-limits based off of the first argument.

```

1 class CustomBucket(Enum):
2     first_arg = 1
3
4     def process(self, *args, **kwargs):
5         if self is CustomBucket.first_arg:
6             # This bucket is based ONLY off
7             # of the first argument passed
8             return args[0]
9
10 # Then to use
11 @cooldown(1, 1, bucket=CustomBucket.first_arg)
12 async def test_func(*args, **kwargs):
13     ....

```

6.9 Stacking cooldown's

Stack as many cooldown's as you want, just note Python starts from the bottom decor and works its way up.

```
1 # Can call ONCE time_period second using the same args
2 # Can call TWICE time_period second using the same kwargs
3 @cooldown(1, 1, bucket=CooldownBucket.args)
4 @cooldown(2, 1, bucket=CooldownBucket.kwargs)
5 async def test_func(*args, **kwargs) -> (tuple, dict):
6     return args, kwargs
```

EXTRA THINGS

`cooldowns.utils.MaybeCoro`

The central part of internal API.

This represents a generic version of type ‘origin’ with type arguments ‘params’. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in `collections.abc`. These must have ‘name’ always set. If ‘inst’ is False, then the alias can’t be instantiated, this is used by e.g. `typing.List` and `typing.Dict`.

alias of `Callable[[Any, Any], Coroutine[Any, Any, Any]]`

`cooldowns.utils.COOLDOWN_ID`

The central part of internal API.

This represents a generic version of type ‘origin’ with type arguments ‘params’. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in `collections.abc`. These must have ‘name’ always set. If ‘inst’ is False, then the alias can’t be instantiated, this is used by e.g. `typing.List` and `typing.Dict`.

alias of `Union[int, str]`

`cooldowns.utils.default_check = <function default_check>`

`class cooldowns.persistence.State(*args, **kwargs)`

`cache: Dict[str, CTPState]`

`cooldown_id: int | str | None`

`limit: int`

`pending_reset: bool`

`time_period: float`

`class cooldowns.persistence.CTPState(*args, **kwargs)`

`current: int`

`limit: int`

`next_reset: List[float]`

`time_period: float`

COOLDOWN REFERENCE

```
class cooldowns.Cooldown(limit: int, time_period: float | ~datetime.timedelta, bucket:
    ~cooldowns.protocols.bucket.CooldownBucketProtocol |
    ~cooldowns.protocols.bucket.AsyncCooldownBucketProtocol | None = None, func:
    ~typing.Callable | None = None, *, cooldown_id: int | str | None = None, check:
    ~typing.Callable[[~typing.Any, ~typing.Any], ~typing.Coroutine[~typing.Any,
    ~typing.Any, ~typing.Any]] | None = <function default_check>)
```

Represents a cooldown for any given **:type:`Callable`**.

```
__init__(limit: int, time_period: float | ~datetime.timedelta, bucket:
    ~cooldowns.protocols.bucket.CooldownBucketProtocol |
    ~cooldowns.protocols.bucket.AsyncCooldownBucketProtocol | None = None, func:
    ~typing.Callable | None = None, *, cooldown_id: int | str | None = None, check:
    ~typing.Callable[[~typing.Any, ~typing.Any], ~typing.Coroutine[~typing.Any, ~typing.Any,
    ~typing.Any]] | None = <function default_check>) → None
```

Parameters

- **limit** (*int*) – How many call's can be made in the time period specified by `time_period`
- **time_period** (*Union[[float](#), [datetime.timedelta](#)]*) – The time period related to limit. This is seconds.
- **bucket** (*Optional[Union[[CooldownBucketProtocol](#), [AsyncCooldownBucketProtocol](#)]]*) – The Bucket implementation to use as a bucket to separate cooldown buckets.
Defaults to `CooldownBucket.all`
- **func** (*Optional[Callable]*) – The function this cooldown is attached to
- **cooldown_id** (*Optional[Union[[int](#), [str](#)]]*) – Useful for resetting individual stacked cooldowns. This should be unique globally, behaviour is not guaranteed if not unique.
- **check** (*Optional[[MaybeCoro](#)]*) – The check used to validate calls to this Cooldown.

This is not used here, however, its required as an implementation detail for shared cooldowns and can be safely ignored as a parameter.

Note: This check will be given the same arguments as the item you are applying the cooldown to.

property bucket: [CooldownBucketProtocol](#) | [AsyncCooldownBucketProtocol](#)

Returns the underlying bucket to process cooldowns against.

clear(*bucket*: [_HashableArguments](#) | *None* = *None*, *, *force_evict*: *bool* = *False*) → *None*

Remove all un-needed buckets, this maintains buckets which are currently tracking cooldown's.

Parameters

- **bucket** (*Optional*[[_HashableArguments](#)]) – The bucket we wish to reset
- **force_evict** (*bool*) – If *True*, delete all tracked cooldown's regardless of whether or not they are needed.

I.e. reset this back to a fresh state.

Notes

You can get [_HashableArguments](#) by using the [Cooldown.get_bucket\(\)](#) method.

property func: [Callable](#) | *None*

Returns the wrapped function.

async get_bucket(**args*, ***kwargs*) → [_HashableArguments](#)

Return the given bucket for some given arguments.

This uses the underlying [CooldownBucket](#) and will return a [_HashableArguments](#) instance which is inline with how [Cooldown](#)'s function.

Parameters

- **args** (*Any*) – The arguments to get a bucket for
- **kwargs** (*Any*) – The keyword arguments to get a bucket for

Returns

An internally correct representation of a bucket for the given arguments.

This can then be used in [Cooldown.clear\(\)](#) calls.

Return type

[_HashableArguments](#)

get_cooldown_times_per(*bucket*: [_HashableArguments](#)) → *TP* | *None*

Return the relevant [CooldownTimesPer](#) object for this bucket, returns *None* if one does not currently exist.

Parameters

bucket ([_HashableArguments](#)) – The bucket you wish to receive against. Get this using [Cooldown.get_bucket\(\)](#)

Returns

The internal [CooldownTimesPer](#) object

Return type

Optional[[CooldownTimesPer](#)]

get_state() → *State*

Return the state of this cooldown as a dictionary in order to be able to persist it.

Returns

This cooldown as a dictionary

Return type

State

load_from_state(state: State) → None

Load this cooldown as per state

Parameters

state (State) – The state you wish to set this cooldown to

Notes

state should be the output of `Cooldown.get_state()` and remain unmodified in order for this operation to work.

async remaining_calls(*args, **kwargs) → int

Given a `:type:`Callable``, return the amount of remaining available calls before these arguments will result in the callable being rate-limited.

Parameters

- **args** –
- **pass.** (Any arguments you will) –
- **kwargs** – Any key-word arguments you will pass.

Returns

How many more times this `:type:`Callable`` can be called without being rate-limited.

Return type

int

```
class cooldowns.StaticCooldown(limit: int, reset_times: ~datetime.time | ~typing.List[~datetime.time], bucket:
    ~cooldowns.protocols.bucket.CooldownBucketProtocol |
    ~cooldowns.protocols.bucket.AsyncCooldownBucketProtocol | None =
    None, func: ~typing.Callable | None = None, *, cooldown_id: int | str |
    None = None, check: ~typing.Callable[[~typing.Any, ~typing.Any],
    ~typing.Coroutine[~typing.Any, ~typing.Any, ~typing.Any]] | None =
    <function default_check>)
```

A cooldown which implements a set amount of cooldown reset times per day.

```
__init__(limit: int, reset_times: ~datetime.time | ~typing.List[~datetime.time], bucket:
    ~cooldowns.protocols.bucket.CooldownBucketProtocol |
    ~cooldowns.protocols.bucket.AsyncCooldownBucketProtocol | None = None, func:
    ~typing.Callable | None = None, *, cooldown_id: int | str | None = None, check:
    ~typing.Callable[[~typing.Any, ~typing.Any], ~typing.Coroutine[~typing.Any, ~typing.Any,
    ~typing.Any]] | None = <function default_check>) → None
```

Parameters

- **limit** (int) – How many call's can be made in the time period specified by time_period
- **reset_times** (Union[datetime.time, List[datetime.time]]) – A time or list of the possible times in the day to reset cooldowns at
- **bucket** (Optional[CooldownBucketProtocol]) – The Bucket implementation to use as a bucket to separate cooldown buckets.

Defaults to `CooldownBucket.all`

- **func** (Optional[Callable]) – The function this cooldown is attached to
- **cooldown_id** (Optional[Union[int, str]]) – Useful for resetting individual stacked cooldowns. This should be unique globally, behaviour is not guaranteed if not unique.

- **check** (*Optional* [*MaybeCoro*]) – The check used to validate calls to this Cooldown.

This is not used here, however, its required as an implementation detail for shared cooldowns and can be safely ignored as a parameter.

Note: This check will be given the same arguments as the item you are applying the cooldown to.

Notes

All times are internally handled based off UTC.

TIMESPER REFERENCE

`class cooldowns.CooldownTimesPer(limit: int, time_period: float, _cooldown: Cooldown)`

`__init__(limit: int, time_period: float, _cooldown: Cooldown) → None`

Parameters

- **limit** (*int*) – How many items are allowed
- **time_period** (*float*) – The period of seconds limit applies to
- **_cooldown** (*Cooldown*) – A backref to the parent cooldown manager.

Notes

This is an internal object. You do not need to construct it yourself.

property fully_reset_at: *datetime* | *None*

When this bucket is fully reset.

Returns

When this bucket fully resets.

Return type

Optional[*datetime.datetime*]

Notes

This will return *None* if it is already fully reset.

property has_cooldown: *bool*

Is this instance currently tracking any cooldowns?

If this returns *False* we can safely delete this instance from the *Cooldown* lookup table.

property next_reset: *datetime* | *None*

When the next window is freed.

Returns

When the next window is freed.

None if there are no windows.

Return type

Optional[*datetime.datetime*]

class `cooldowns.StaticTimesPer`(*limit*: `int`, *reset_times*: `List[datetime.time]`, *_cooldown*: `Cooldown`)

A cooldown which implements a set amount of cooldown reset times.

__init__(*limit*: `int`, *reset_times*: `List[datetime.time]`, *_cooldown*: `Cooldown`) → `None`

Parameters

- **limit** (`int`) – How many items are allowed
- **reset_times** (`List[datetime.time]`) – A list of the possible times in the day to reset cooldowns at
- **_cooldown** (`Cooldown`) – A backref to the parent cooldown manager.

Notes

This is an internal object. You do not need to construct it yourself.

get_next_reset(*now*: `datetime`) → `datetime`

Fetches the next possible reset.

static next_datetime(*current*: `datetime`, *time*: `time`) → `datetime`

PROTOCOL REFERENCE

The protocol to implement to fit the needs of a Bucket

class cooldowns.protocols.bucket.**CooldownBucketProtocol**(*args, **kwargs)

CooldownBucketProtocol implementation Protocol.

process(*args, **kwargs) → *Any*

Returns

The values returned from this method will be used to represent a bucket.

Return type

Any

class cooldowns.protocols.bucket.**AsyncCooldownBucketProtocol**(*args, **kwargs)

AsyncCooldownBucketProtocol implementation Protocol.

async process(*args, **kwargs) → *Any*

Returns

The values returned from this method will be used to represent a bucket.

Return type

Any

HASHABLEARGUMENTS REFERENCE

An implementation detail.

```
class cooldowns.buckets._HashableArguments(*args, **kwargs)
```

An implementation class, you don't need to create these yourself.

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

Symbols

`_HashableArguments` (class in `cooldowns.buckets`), 35
`__init__()` (`cooldowns.Cooldown` method), 27
`__init__()` (`cooldowns.CooldownTimesPer` method), 31
`__init__()` (`cooldowns.StaticCooldown` method), 29
`__init__()` (`cooldowns.StaticTimesPer` method), 32

A

`AsyncCooldownBucketProtocol` (class in `cooldowns.protocols.bucket`), 33

B

`BaseCooldownException` (class in `cooldowns.exceptions`), 11
`bucket` (`cooldowns.Cooldown` property), 27

C

`cache` (`cooldowns.persistence.State` attribute), 25
`CallableOnCooldown` (class in `cooldowns.exceptions`), 11
`clear()` (`cooldowns.Cooldown` method), 27
`Cooldown` (class in `cooldowns`), 27
`cooldown` (`cooldowns.exceptions.CallableOnCooldown` attribute), 11
`cooldown()` (in module `cooldowns`), 3
`cooldown_id` (`cooldowns.persistence.State` attribute), 25
`COOLDOWN_ID` (in module `cooldowns.utils`), 25
`CooldownAlreadyExists` (class in `cooldowns.exceptions`), 11
`CooldownBucketProtocol` (class in `cooldowns.protocols.bucket`), 33
`CooldownTimesPer` (class in `cooldowns`), 31
`CTPState` (class in `cooldowns.persistence`), 25
`current` (`cooldowns.persistence.CTPState` attribute), 25

D

`default_check` (in module `cooldowns.utils`), 25
`define_shared_cooldown()` (in module `cooldowns`), 5
`define_shared_static_cooldown()` (in module `cooldowns`), 5

F

`fully_reset_at` (`cooldowns.CooldownTimesPer` property), 31
`func` (`cooldowns.Cooldown` property), 28
`func` (`cooldowns.exceptions.CallableOnCooldown` attribute), 11

G

`get_all_cooldowns()` (in module `cooldowns`), 7
`get_bucket()` (`cooldowns.Cooldown` method), 28
`get_cooldown()` (in module `cooldowns`), 7
`get_cooldown_times_per()` (`cooldowns.Cooldown` method), 28
`get_next_reset()` (`cooldowns.StaticTimesPer` method), 32
`get_remaining_calls()` (in module `cooldowns`), 6
`get_shared_cooldown()` (in module `cooldowns`), 7
`get_state()` (`cooldowns.Cooldown` method), 28

H

`has_cooldown` (`cooldowns.CooldownTimesPer` property), 31

L

`limit` (`cooldowns.persistence.CTPState` attribute), 25
`limit` (`cooldowns.persistence.State` attribute), 25
`load_from_state()` (`cooldowns.Cooldown` method), 28

M

`MaybeCoro` (in module `cooldowns.utils`), 25

N

`next_datetime()` (`cooldowns.StaticTimesPer` static method), 32
`next_reset` (`cooldowns.CooldownTimesPer` property), 31
`next_reset` (`cooldowns.persistence.CTPState` attribute), 25
`NonExistent` (class in `cooldowns.exceptions`), 11
`NoRegisteredCooldowns` (class in `cooldowns.exceptions`), 11

P

`pending_reset` (*cooldowns.persistence.State* attribute), 25
`process()` (*cooldowns.protocols.bucket.AsyncCooldownBucketProtocol* method), 33
`process()` (*cooldowns.protocols.bucket.CooldownBucketProtocol* method), 33

R

`remaining_calls()` (*cooldowns.Cooldown* method), 29
`reset_bucket()` (*in module cooldowns*), 7
`reset_cooldown()` (*in module cooldowns*), 6
`reset_cooldowns()` (*in module cooldowns*), 6
`resets_at` (*cooldowns.exceptions.CallableOnCooldown* attribute), 11
`retry_after` (*cooldowns.exceptions.CallableOnCooldown* property), 11

S

`shared_cooldown()` (*in module cooldowns*), 4
State (class *in cooldowns.persistence*), 25
`static_cooldown()` (*in module cooldowns*), 3
StaticCooldown (class *in cooldowns*), 29
StaticTimesPer (class *in cooldowns*), 31

T

`time_period` (*cooldowns.persistence.CTPState* attribute), 25
`time_period` (*cooldowns.persistence.State* attribute), 25

U

UnknownBucket (class *in cooldowns.exceptions*), 11