Function Cooldowns

Skelmis

Oct 22, 2022

MAIN:

Cooldowns for Coroutines, simple as that.

CHAPTER

ONE

DECORATOR USAGE

You have two choices for decorators, as documented below.

For example usages, view the examples section.

cooldowns.**cooldown**(*limit: int, time_period: ~typing.Union[float, ~datetime.timedelta], bucket: ~cooldowns.protocols.bucket.CooldownBucketProtocol, check: ~typing.Optional[~typing.Callable[[~typing.Any, ~typing.Any], ~typing.Coroutine[~typing.Any, ~typing.Any, ~typing.Any]]] = <function default_check>, *, cooldown_id: ~typing.Optional[~typing.Union[int, str]] = 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** (`CooldownBucketProtocol`) – 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.**shared_cooldown**(*cooldown_id: Optional[Union[int, str]]*)

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.

CHAPTER

TWO

UTILITY USAGE

Various utilities for using cooldowns.

cooldowns.**define_shared_cooldown**(*limit: int*, *time_period: float*, *bucket: CooldownBucketProtocol*, *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** (`CooldownBucketProtocol`) – 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.

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: Union[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.

cooldowns.**reset_bucket**(*func: Callable[[Any, Any], Coroutine[Any, 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*) → *Cooldown*
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*) → *Cooldown*

Retrieve a shared *Cooldown* 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

CHAPTER

THREE

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.

CHAPTER

FOUR

LIBRARY EXCEPTIONS

All exceptions inherit from `BaseCooldownException`

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

A base exception handler.

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

There doesnt 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*

CHAPTER

FIVE

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.

```
import cooldowns

...

@bot.slash_command(
    description="Ping command",
)
@cooldowns.cooldown(1, 15, bucket=cooldowns.SlashBucket.author)
async def ping(interaction: nextcord.Interaction):
    await interaction.response.send_message("Pong!")
```

5.2 Example cog usage

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

```
import cooldowns

...

@nextcord.slash_command(
    description="Ping command",
)
@cooldowns.cooldown(1, 30, bucket=cooldowns.SlashBucket.guild)
async def ping(self, interaction: nextcord.Interaction):
    await interaction.response.send_message("Pong!")
```

5.3 Handling cooldown's

Here is an example error handler

```
from cooldowns import CallableOnCooldown

...

@bot.event
async def on_application_command_error(inter: nextcord.Interaction, error):
    error = getattr(error, "original", error)

    if isinstance(error, CallableOnCooldown):
        await inter.send(
            f"You are being rate-limited! Retry in `{error.retry_after}` seconds."
        )

    else:
        raise error
```

5.4 Get Remaining Calls

```
from cooldowns import get_remaining_calls, cooldown, SlashBucket

@bot.slash_command()
@cooldown(2, 10, SlashBucket.command)
async def test(inter):
    ...
    calls_left = get_remaining_calls(test, inter)
    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.

```
from cooldowns import cooldown, CooldownBucket, reset_cooldown

@cooldown(1, 30, CooldownBucket.all, cooldown_id="my_cooldown")
async def test(*args, **kwargs):
    ...

# Reset
reset_cooldown("my_cooldown")
```

5.6 Reset all cooldowns on a callable

Resets all cooldowns on the provided callable.

```
from cooldowns import cooldown, CooldownBucket, reset_cooldowns

@cooldown(1, 30, CooldownBucket.all)
@cooldown(1, 15, CooldownBucket.args)
async def test(*args, **kwargs):
    ...

# Reset
reset_cooldowns(test)
```

5.7 Reset a specific bucket on a cooldown

Resets only the given buckets on a cooldown.

```
from cooldowns import cooldown, CooldownBucket, reset_bucket

@cooldown(1, 30, CooldownBucket.all)
async def test(*args, **kwargs):
    ...

...

# Reset the bucket with `1` as an argument
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.

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

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

```
async def mock_db_check(*args, **kwargs):
    # You can do database calls here or anything
    # since this is an async context
    return args[0] == 1

@cooldown(1, 1, bucket=CooldownBucket.args, check=mock_db_check)
async def test_func(*args, **kwargs) -> (tuple, dict):
    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.

```
class CustomBucket(Enum):
    first_arg = 1

    def process(self, *args, **kwargs):
        if self is CustomBucket.first_arg:
            # This bucket is based ONLY off
            # of the first argument passed
            return args[0]

# Then to use
@cooldown(1, 1, bucket=CustomBucket.first_arg)
async def test_func(*args, **kwargs):
    .....
```

5.10 Stacking cooldown's

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

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

5.11 Shared cooldowns

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

```
from cooldowns import define_shared_cooldown, shared_cooldown, CooldownBucket

define_shared_cooldown(1, 5, CooldownBucket.all, cooldown_id="my_id")

@shared_cooldown("my_id")
async def test_1(*args, **kwargs):
    return 1

@shared_cooldown("my_id")
async def test_2(*args, **kwargs):
    return 2

# These now both share the same cooldown
```

5.12 Manually using cooldowns

How to use the Cooldown object without a decorator.

```
from cooldowns import Cooldown, CooldownBucket

cooldown = Cooldown(1, 5, CooldownBucket.args)

async with cooldown(*args, **kwargs):
    # This will apply the cooldown
    ...
    # Do things
```

CHAPTER

SIX

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

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

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.

```
from cooldowns import CooldownBucket

CooldownBucket.all
CooldownBucket.args
CooldownBucket.kwargs

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

Attributes
==========
all
    The buckets are defined using all
    arguments passed to the :type:`Callable`
args
    The buckets are defined using all
    non-keyword arguments passed to the :type:`Callable`
kwargs
    The buckets are defined using all
    keyword arguments passed to the :type:`Callable`
"""
```

Slash command bucket.

```
from cooldowns import SlashBucket

SlashBucket.author
SlashBucket.guild
SlashBucket.channel
SlashBucket.command

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

Attributes
==========
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.
"""
```

6.3 Example usage

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

```
import cooldowns

...

@bot.slash_command(
    description="Ping command",
)
@cooldowns.cooldown(1, 15, bucket=cooldowns.SlashBucket.author)
async def ping(interaction: nextcord.Interaction):
    await interaction.response.send_message("Pong!")
```

6.4 Example cog usage

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

```
import cooldowns

...

@nextcord.slash_command(
    description="Ping command",
)
@cooldowns.cooldown(1, 30, bucket=cooldowns.SlashBucket.guild)
async def ping(self, interaction: nextcord.Interaction):
    await interaction.response.send_message("Pong!")
```

6.5 Handling cooldown's

Here is an example error handler

```
from cooldowns import CallableOnCooldown

...

@bot.event
async def on_application_command_error(inter: nextcord.Interaction, error):
    error = getattr(error, "original", error)

    if isinstance(error, CallableOnCooldown):
        await inter.send(
            f"You are being rate-limited! Retry in `{error.retry_after}` seconds."
        )

    else:
        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

```
def get_remaining_calls(func: MaybeCoro, *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
    ----------
    func: MaybeCoro
        The :type:`Callable` you want to check.
    args
        Any arguments you will pass.
    kwargs
        Any key-word arguments you will pass.

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

    Raises
    ------
    NoRegisteredCooldowns
        The given :type:`Callable` has no cooldowns.

    Notes
    -----
    This aggregates all attached cooldowns
    and returns the lowest remaining amount.
    """
```

Example usage

```
from cooldowns import get_remaining_calls, cooldown, SlashBucket

```

(continues on next page)

(continued from previous page)

```
@bot.slash_command()
@cooldown(2, 10, SlashBucket.command)
async def test(inter):
    ...
    calls_left = get_remaining_calls(test, inter)
    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.

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

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

```
async def mock_db_check(*args, **kwargs):
    # You can do database calls here or anything
    # since this is an async context
    return args[0] == 1

@cooldown(1, 1, bucket=CooldownBucket.args, check=mock_db_check)
async def test_func(*args, **kwargs) -> (tuple, dict):
    return args, kwargs
```

6.8 Custom buckets

All you need is an enum with the `process` method.

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

```
class CustomBucket(Enum):
    first_arg = 1

    def process(self, *args, **kwargs):
        if self is CustomBucket.first_arg:
            # This bucket is based ONLY off
            # of the first argument passed
            return args[0]

# Then to use
@cooldown(1, 1, bucket=CustomBucket.first_arg)
async def test_func(*args, **kwargs):
    .....
```

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.

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

CHAPTER

SEVEN

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

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`

CHAPTER

EIGHT

COOLDOWN REFERENCE

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

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

__init__(*limit: int, time_period: ~typing.Union[float, ~datetime.timedelta], bucket: ~typing.Optional[~cooldowns.protocols.bucket.CooldownBucketProtocol] = None, func: ~typing.Optional[~typing.Callable] = None, *, cooldown_id: ~typing.Optional[~typing.Union[int, str]] = None, check: ~typing.Optional[~typing.Callable[[~typing.Any, ~typing.Any], ~typing.Coroutine[~typing.Any, ~typing.Any, ~typing.Any]]] = <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[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.

---

property **bucket**: *CooldownBucketProtocol*

Returns the underlying bucket to process cooldowns against.

clear(*bucket: Optional[_HashableArguments] = 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: Optional[Callable]

Returns the wrapped function.

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) → Optional[*CooldownTimesPer*]

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.

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

CHAPTER

NINE

COOLDOWNTIMESPER 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: Optional[datetime]

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: Optional[datetime]

When the next window is freed.

Returns

When the next window is freed.

None if there are no windows.

Return type
Optional[datetime.datetime]

CHAPTER

TEN

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

CHAPTER

ELEVEN

HASHABLEARGUMENTS REFERENCE

An implementation detail.

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

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

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

B

C

D

F

G

H

L

M

N

P

R

S

T

U