riprova

Release 0.3.0

May 20, 2023

Contents

`riprova` (meaning `retry` in Italian) is a small, general-purpose and versatile Python library that provides retry mechanisms with multiple backoff strategies for any sort of failed operations.

It's domain agnostic, highly customizable, extensible and provides a minimal API that's easy to instrument in any code base via decorators, context managers or raw API consumption.

For a brief introduction about backoff mechanisms for potential failed operations, read this article.

CHAPTER 1

Features

- Retry decorator for simple and idiomatic consumption.
- Simple Pythonic programmatic interface.
- Maximum retry timeout support.
- Supports error whitelisting and blacklisting.
- Supports custom error evaluation retry logic (useful to retry only in specific cases).
- Automatically retry operations on raised exceptions.
- Supports asynchronous coroutines with both `async/await` and `yield from` syntax.
- Configurable maximum number of retry attempts.
- Highly configurable supporting max retries, timeouts or retry notifier callback.
- Built-in backoff strategies: constant, fibonacci and exponential backoffs.
- Supports sync/async context managers.
- Pluggable custom backoff strategies.
- Lightweight library with almost zero embedding cost.
- Works with Python +2.6, 3.0+ and PyPy.

CHAPTER 2

Backoff strategies

List of built-in backoff strategies.

- Constant backoff
- Fibonacci backoff
- Exponential backoff

You can also implement your own one easily. See ConstantBackoff for an implementation reference.

CHAPTER 3

Installation

Using `pip` package manager (requires pip 1.9+. Upgrade it running: `pip install -U pip`):

```
pip install -U riprova
```

Or install the latest sources from Github:

```
pip install -e git+git://github.com/h2non/riprova.git#egg=riprova
```

CHAPTER 4

API

- riprova.retry
- riprova.Retrier
- riprova.AsyncRetrier
- riprova.Backoff
- riprova.ConstantBackoff
- riprova.FibonacciBackoff
- riprova.ExponentialBackoff
- riprova.ErrorWhitelist
- riprova.ErrorBlacklist
- riprova.add_whitelist_error
- riprova.RetryError
- riprova.RetryTimeoutError
- riprova.MaxRetriesExceeded
- riprova.NotRetriableError

4.1 Examples

You can see more featured examples from the *documentation* site.

Basic usage examples:

```
import riprova

@riprova.retry
```

(continues on next page)

(continued from previous page)

```
def task():
    """Retry operation if it fails with constant backoff (default)"""

@riprova.retry(backoff=riprova.ConstantBackoff(retries=5))
def task():
    """Retry operation if it fails with custom max number of retry attempts"""

@riprova.retry(backoff=riprova.ExponentialBackOff(factor=0.5))
def task():
    """Retry operation if it fails using exponential backoff"""

@riprova.retry(timeout=10)
def task():
    """Raises a TimeoutError if the retry loop exceeds from 10 seconds"""

def on_retry(err, next_try):
    print('Operation error: {}'.format(err))
    print('Next try in: {}ms'.format(next_try))

@riprova.retry(on_retry=on_retry)
def task():
    """Subscribe via function callback to every retry attempt"""

def evaluator(response):
    # Force retry operation if not a valid response
    if response.status >= 400:
        raise RuntimeError('invalid response status')  # or simple return True
    # Otherwise return False, meaning no retry
    return False

@riprova.retry(evaluator=evaluator)
def task():
    """Use a custom evaluator function to determine if the operation failed or not"""

@riprova.retry
async def task():
    """Asynchronous coroutines are also supported :)"""
```

Retry failed HTTP requests:

```
import pook
import requests
from riprova import retry

# Define HTTP mocks to simulate failed requests
pook.get('server.com').times(3).reply(503)
pook.get('server.com').times(1).reply(200).json({'hello': 'world'})


# Retry evaluator function used to determine if the operated failed or not
def evaluator(response):
    if response != 200:
        return Exception('failed request')  # you can also simply return True
    return False


# On retry even subscriptor
```

(continues on next page)

(continued from previous page)

```
def on_retry(err, next_try):
    print('Operation error {}'.format(err))
    print('Next try in {}ms'.format(next_try))


# Register retriable operation
@retry(evaluator=evaluator, on_retry=on_retry)
def fetch(url):
    return requests.get(url)


# Run task that might fail
fetch('http://server.com')
```

CHAPTER 5

License

MIT - Tomas Aparicio

CHAPTER 6

Contents

6.1 Examples

6.1.1 Generic showcase examples

```
# flake8: noqa
import riprova

@riprova.retry
def task():
    """Retry operation if it fails with constant backoff"""

@riprova.retry(backoff=riprova.ExponentialBackOff(factor=.5))
def task():
    """Retry operation if it fails using exponential backoff"""

@riprova.retry(timeout=10)
def task():
    """Raises a TimeoutError if the retry loop exceeds from 10 seconds"""

def on_retry(err, next_try):
    print('Operation error: {}'.format(err))
    print('Next try in: {}ms'.format(next_try))

@riprova.retry(on_retry=on_retry)
def task():
    """Subscribe via function callback to every retry attempt"""

def evaluator(response):
    # Force retry operation if not a valid response
    if response.status >= 400:
        raise RuntimeError('invalid response status')
    # Otherwise return False, meaning no retry
```

(continues on next page)

(continued from previous page)

```
    return False

@riprova.retry(evaluator=evaluator)
def task():
    """Use a custom evaluator function to determine if the operation failed or not"""

@riprova.retry
async def task():
    """Asynchronous coroutines are also supported :)"""
```

6.1.2 Usage as decorator

```
# -*- coding: utf-8 -*-
import riprova

# Store number of function calls for error simulation
calls = 0


# Register retriable operation with custom evaluator
@riprova.retry
def mul2(x):
    global calls

    if calls < 3:
        calls += 1
        raise RuntimeError('simulated call error')

    return x * 2


# Run task
result = mul2(2)
print('Result: {}'.format(result))
```

6.1.3 Usage as context manager

```
# -*- coding: utf-8 -*-
import riprova

# Store number of function calls for error simulation
calls = 0


# Register retriable operation with custom evaluator
def mul2(x):
    global calls

    if calls < 4:
        calls += 1
        raise RuntimeError('simulated call error')
```

(continues on next page)

(continued from previous page)

```
    return x * 2


# Run task via context manager
with riprova.Retrier() as retry:
    result = retry.run(mul2, 2)
    print('Result 1: {}'.format(result))


# Or alternatively create a shared retrier and reuse it across multiple
# context managers.
retrier = riprova.Retrier()

with retrier as retry:
    calls = 0
    result = retry.run(mul2, 4)
    print('Result 2: {}'.format(result))

with retrier as retry:
    calls = 0
    result = retry.run(mul2, 8)
    print('Result 3: {}'.format(result))
```

6.1.4 Timeout retry cycle limit

```
# -*- coding: utf-8 -*-
import riprova

# Store number of function calls for error simulation
calls = 0


# Register retriable operation with custom evaluator
@riprova.retry(timeout=0.3)
def mul2(x):
    global calls

    if calls < 4:
        calls += 1
        raise RuntimeError('simulated call error')

    return x * 2


# Run task
try:
    mul2(2)
except riprova.RetryTimeoutError as err:
    print('Timeout error: {}'.format(err))
```

6.1.5 Retry failed HTTP requests

```
# -*- coding: utf-8 -*-
import pook
import requests
from riprova import retry

# Define HTTP mocks
pook.get('server.com').times(3).reply(503)
pook.get('server.com').times(1).reply(200).json({'hello': 'world'})


# Retry evaluator function used to determine if the operated failed or not
def evaluator(response):
    if response.status != 200:
        return Exception('failed request')
    return False


# On retry even subcriptor
def on_retry(err, next_try):
    print('Operation error {}'.format(err))
    print('Next try in {}ms'.format(next_try))


# Register retriable operation
@retry(evaluator=evaluator, on_retry=on_retry)
def fetch(url):
    return requests.get(url)


# Run request
fetch('http://server.com')
```

6.1.6 Retry failed HTTP requests using asyncio + aiohttp

```
# -*- coding: utf-8 -*-
# flake8: noqa
"""
Note: only Python 3.5+ compatible.
"""
import pook
import paco
import aiohttp
from riprova import retry

# Define HTTP mocks to simulate failed scenarios
pook.get('server.com').times(4).reply(503)
pook.get('server.com').times(1).reply(200).json({'hello': 'world'})


# Retry evaluator function used to determine if the operated failed or not
async def evaluator(status):
    if status != 200:
        return Exception('failed request with status {}'.format(status))
```

(continues on next page)

(continued from previous page)

```
    return False


# On retry even subcriptor
async def on_retry(err, next_try):
    print('Operation error: {}'.format(err))
    print('Next try in {}ms'.format(next_try))


# Register retriable operation with custom evaluator
@retry(evaluator=evaluator, on_retry=on_retry)
async def fetch(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return response.status


# Run request
status = paco.run(fetch('http://server.com'))
print('Response status:', status)
```

6.1.7 Whitelisting custom errors

```
# -*- coding: utf-8 -*-
import riprova


# Custom error object
class MyCustomError(Exception):
    pass


# Whitelist of errors that should not be retried
whitelist = riprova.ErrorWhitelist([
    ReferenceError,
    ImportError,
    IOError,
    SyntaxError,
    IndexError
])


def error_evaluator(error):
    """
    Used to determine if an error is legit and therefore
    should be retried or not.
    """
    return whitelist.isretry(error)


# In order to define a global whitelist policy that would be used
# across all retry instances, overwrite the whitelist attribute in Retrier:
riprova.Retrier.whitelist = whitelist

# Store number of function calls for error simulation
```

(continues on next page)

(continued from previous page)

```
calls = 0


# Register retriable operation with a custom error evaluator
# You should pass the evaluator per retry instance.
@riprova.retry(error_evaluator=error_evaluator)
def mul2(x):
    global calls

    if calls < 3:
        calls += 1
        raise RuntimeError('simulated call error')

    if calls == 3:
        calls += 1
        raise ReferenceError('legit error')

    return x * 2


# Run task
try:
    mul2(2)
except ReferenceError as err:
    print('Whitelisted error: {}'.format(err))
    print('Retry attempts: {}'.format(calls))
```

6.1.8 Blacklisting custom errors

```
# -*- coding: utf-8 -*-
import riprova


# Custom error object
class MyCustomError(Exception):
    pass


# Blacklist of errors that should exclusively be retried
blacklist = riprova.ErrorBlacklist([
    RuntimeError,
    MyCustomError
])


def error_evaluator(error):
    """
    Used to determine if an error is legit and therefore
    should be retried or not.
    """
    return blacklist.isretry(error)


# In order to define a global blacklist policy that would be used
# across all retry instances, overwrite the blacklist attribute in Retrier.
```

(continues on next page)

(continued from previous page)

```
# NOTE: blacklist overwrites any whitelist. They are mutually exclusive.
riprova.Retrier.blacklist = blacklist

# Store number of function calls for error simulation
calls = 0


# Register retriable operation with a custom error evaluator
# You should pass the evaluator per retry instance.
@riprova.retry(error_evaluator=error_evaluator)
def mul2(x):
    global calls

    if calls < 3:
        calls += 1
        raise RuntimeError('simulated call error')

    if calls == 3:
        calls += 1
        raise Exception('non blacklisted error')

    return x * 2


# Run task
try:
    mul2(2)
except Exception as err:
    print('Blacklisted error: {}'.format(err))
    print('Retry attempts: {}'.format(calls))
```

6.1.9 Constant backoff strategy

```
# -*- coding: utf-8 -*-
import riprova

# Store number of function calls for error simulation
calls = 0

# Max number of retries attempts
retries = 5


# Register retriable operation with custom evaluator
@riprova.retry(backoff=riprova.ConstantBackoff(interval=.5, retries=retries))
def mul2(x):
    global calls

    if calls < 4:
        calls += 1
        raise RuntimeError('simulated call error')

    return x * 2
```

(continues on next page)

(continued from previous page)

```
# Run task
result = mul2(2)
print('Result: {}'.format(result))
```

6.1.10 Fibonacci backoff strategy

```
# -*- coding: utf-8 -*-
import riprova

# Store number of function calls for error simulation
calls = 0

# Max number of retries attempts
retries = 5


# Register retriable operation with custom evaluator
@riprova.retry(backoff=riprova.FibonacciBackoff(retries=retries))
def mul2(x):
    global calls

    if calls < 4:
        calls += 1
        raise RuntimeError('simulated call error')

    return x * 2


# Run task
result = mul2(2)
print('Result: {}'.format(result))
```

6.1.11 Exponential backoff strategy

```
# -*- coding: utf-8 -*-
import riprova

# Store number of function calls for error simulation
calls = 0

# Max number of retries attempts
retries = 5


# Register retriable operation with custom evaluator
@riprova.retry(backoff=riprova.ExponentialBackOff(factor=1, retries=retries))
def mul2(x):
    global calls

    if calls < 4:
        calls += 1
        raise RuntimeError('simulated call error')
```

(continues on next page)

(continued from previous page)

```
    return x * 2


# Run task
result = mul2(2)
print('Result: {}'.format(result))
```

6.2 API documentation

riprova.**retry**(*timeout=0*, *backoff=None*, *evaluator=None*, *error_evaluator=None*, *on_retry=None*, ***kw*)

Decorator function that wraps function, method or coroutine function that would be retried on failure capabilities.

Retry policy can be configured via *backoff* param.

You can also use a custom evaluator function used to determine when the returned task value is valid or not, retrying the operation accordingly.

You can subscribe to every retry attempt via *on_retry* param, which accepts a function or a coroutine function.

This function is overloaded: you can pass a function or coroutine function as first argument or an *int* indicating the *timeout* param.

This function as decorator.

Parameters

- **timeout** (*int*) – optional maximum timeout in seconds. Use *0* for no limit. Defaults to *0*.
- **backoff** (riprova.Backoff) – optional backoff strategy to use. Defaults to *riprova.ConstantBackoff*.
- **evaluator** (*function|coroutinefunction*) – optional retry result evaluator function used to determine if an operator failed or not. Useful when domain-specific evaluation, such as valid HTTP responses.
- **error_evaluator** (*function|coroutinefunction*) – optional error evaluator function usef to determine if a reased exception is legit or not, and therefore should be handled as a failure or simply forward the raised exception and stop the retry cycle. This is useful in order to ignore custom error exceptions.
- **on_retry** (*function|coroutinefunction*) – optional on retry event subscriber that will be executed before every retry attempt. Useful for reporting and tracing.
- **sleep_fn** (*function|coroutinefunction*) – optional sleep function to be used before retry attempts. Defaults to *time.sleep()* or *asyncio.sleep()*.
- ***kwargs** (*mixed*) – keyword variadic arguments to pass to *Retrier* or *AsyncRetrier* class constructors.

Raises TypeError – if function is not a function or coroutine function.

Returns

decorated function or coroutine function with retry mechanism.

Return type function or coroutinefunction

Usage:

```
@riprova.retry
def task(x, y):
    return x * y

task(4, 4)
# => 16

@riprova.retry(backoff=riprova.FinonacciBackoff(retries=10))
def task(x, y):
    return x * y

task(4, 4)
# => 16

@riprova.retry(timeout=10)
async def task(x, y):
    return x * y

await task(4, 4)
# => 16

def on_retry(err, next_try):
    print('Error exception: {}'.format(err))
    print('Next try in {}ms'.format(next_try))

@riprova.retry(on_retry=on_retry)
async def task(x, y):
    return x * y

await task(4, 4)
# => 16
```

class riprova.**Retrier**(*timeout=0*, *backoff=None*, *evaluator=None*, *error_evaluator=None*, *on_retry=None*, *sleep_fn=None*)

Bases: object

Implements a simple function retry mechanism with configurable backoff strategy and task timeout limit handler.

Additionally, you can subscribe to *retry* attempts via *on_retry* param, which accepts a binary function.

Retrier object also implements a context manager.

Parameters

- **timeout** (*int*) – maximum optional timeout in milliseconds. Use *0* for no limit. Defaults to *0*.
- **backoff** (riprova.Backoff) – optional backoff strategy to use. Defaults to *riprova.ConstantBackoff*.
- **evaluator** (*function*) – optional evaluator function used to determine when an operation should be retried or not. This allow the developer to retry operations that do not raised any exception, for instance. Evaluator function accepts 1 argument: the returned task result. Evaluator function can raise an exception, return an error or simply return *True* in order to retry the operation. Otherwise the operation will be considered as valid and the retry loop will end.
- **error_evaluator** (*function*) – optional evaluator function used to determine when a task raised exception should be proccesed as legit error and therefore retried or, otherwise,

treated as whitelist error, stopping the retry loop and re-raising the exception to the task consumer. This provides high versatility to developers in order to compose any exception, for instance. Evaluator is an unary function that accepts 1 argument: the raised exception object. Evaluator function can raise an exception, return an error or simply return *True* in order to retry the operation. Otherwise the operation will be considered as valid and the retry loop will end.

- **on_retry** (*function*) – optional function to call on before very retry operation. *on_retry* function accepts 2 arguments: *err, next_try* and should return nothing.
- **sleep_fn** (*function*) – optional function used to sleep. Defaults *time.sleep()*.

whitelist
default error whitelist instance used to evaluate when.

Type *riprova.ErrorWhitelist*

blacklist
default error blacklist instance used to evaluate when. Blacklist and Whitelist are mutually exclusive.

Type *riprova.ErrorBlacklist*

timeout
stores the maximum retries attempts timeout in seconds. Use *0* for no limit. Defaults to *0*.

Type int

attempts
number of retry attempts being executed from last *run()* method call.

Type int

error
stores the latest generated error. *None* if not error yet from last *run()* execution.

Type Exception

sleep
stores the function used to sleep. Defaults to *time.sleep*.

Type function

backoff
stores current used backoff. Defaults to *riprova.ConstantBackoff*.

Type *Backoff*

evaluator
stores the used evaluator function. Defaults to *None*.

Type function

error_evaluator
stores the used error evaluator function. Defaults to *self.is_whitelisted_error()*.

Type function

on_retry
stores the retry notifier function. Defaults to *None*.

Type function

Raises AssertionError – in case of invalid input params.

Usage:

```
# Basic usage
retrier = riprova.Retrier(
    timeout=10 * 1000,
    backoff=riprova.FibonacciBackoff(retries=5))

def task(x):
    return x * x

result = retrier.run(task, 4)
assert result == 16
assert retrier.attempts == 0
assert retrier.error == None

# Using the retrier
retrier = riprova.Retrier(
    timeout=10 * 1000,
    backoff=riprova.FibonacciBackoff(retries=5))

def task(x):
    return x * x

result = retrier.run(task, 4)
assert result == 16
assert retrier.attempts == 0
assert retrier.error == None

# Using the context manager
with riprova.Retrier() as retry:
    retry.run(task, 'foo', bar=1)
```

blacklist = None

is_whitelisted_error(*err*)

istimeout(*start*)
Verifies if the current timeout.

Parameters **start** (*int*) – start UNIX time in milliseconds.

Returns *True* if timeout exceeded, otherwise *False*.

Return type bool

run(*fn*, **args*, ***kw*)
Runs the given function in a retry loop until the operation is completed successfully or maximum retries attempts are reached.

Parameters

- **fn** (*function*) – operation to retry.
- ***args** (*args*) – partial arguments to pass to the function.
- ***kw** (*kwargs*) – partial keyword arguments to pass to the function.

Raises

- Exception – any potential exception raised by the function.
- *RetryTimeoutError* – in case of a timeout exceed.
- RuntimeError – if evaluator function returns *True*.

Returns value returned by the original function.

Return type mixed

whitelist = None

class riprova.**Backoff**

Bases: object

Backoff representing the minimum implementable interface by backoff strategies.

This class does not provide any logic, it's simply used for documentation purposes and type polymorphism.

Backoff implementations are intended to be used in a single-thread context.

STOP = -1

next()

Returns the number of seconds to wait before the next try, otherwise returns *Backoff.STOP*, which indicates the max number of retry operations were reached.

Backoff strategies must implement this method.

Returns time to wait in seconds before the next try.

Return type int

reset()

Resets the current backoff state data.

Backoff strategies must implement this method.

class riprova.**ConstantBackoff**(*interval=0.1*, *retries=10*)

Bases: riprova.backoff.Backoff

ConstantBackOff is a backoff policy that always returns the same backoff delay.

The constant backoff policy can be configured with a custom retry interval and maximum number of retries.

This is in contrast to an exponential backoff policy, which returns a delay that grows longer as you call *next()*.

ConstantBackoff is expected to run in a single-thread context.

Parameters

- **interval** (*int | float*) – wait interval before retry in seconds. Use *0* for no wait. Defaults to *0.1 = 100* milliseconds.
- **retries** (*int*) – maximum number of retries attempts. Use *0* for no limit. Defaults to *10*.

Raises AssertionError – in case of invalid params.

Usge:

```
@riprova.retry(backoff=riprova.ConstantBackoff(retries=5))
def task(x):
    return x * x
```

next()

Returns the number of seconds to wait before the next try, otherwise returns *Backoff.STOP*, which indicates the max number of retry operations were reached.

Returns time to wait in seconds before the next try.

Return type float

reset()
Resets the current backoff state data.

class riprova.**FibonacciBackoff**(*retries=10*, *initial=1*, *multiplier=1*)
Bases: riprova.backoff.Backoff

Implements a backoff policy based on Fibonacci sequence of numbers.

The Fibonacci backoff policy can be configured with a custom initial sequence number value and maximum number of retries.

This policy is similar to exponential backoff policy, which returns a delay that grows longer as you call *next()*.

For more information, see: https://en.wikipedia.org/wiki/Fibonacci_number

FibonacciBackoff is expected to run in a single-thread context.

Parameters

- **retries** (*int*) – maximum number of retries. Use *0* for no limit. Defaults to *10*.
- **initial** (*int*) – initial number for fibonacci series. Detaults to *1*.
- **multiper** (*int*) – fibonacci series number time multiplier. Defaults to *1*.

Raises AssertionError – in case of invalid params.

Usage:

```
@riprova.retry(backoff=riprova.FibonacciBackoff(retries=5))
def task(x):
    return x * x
```

interval
Returns the next Fibonacci number in the series, multiplied by the current configured multiplier, typically *100*, for time seconds adjust.

next()
Returns the number of seconds to wait before the next try, otherwise returns *Backoff.STOP*, which indicates the max number of retry operations were reached.

Returns time to wait in seconds before the next try.

Return type float

reset()
Resets the current backoff state data.

class riprova.**ExponentialBackOff**(*interval=0.5*, *factor=0.5*, *max_interval=60*, *max_elapsed=900*, *multiplier=1.5*)
Bases: riprova.backoff.Backoff

ExponentialBackOff is a backoff implementation that increases the backoff period for each retry attempt using a randomization function that grows exponentially.

next() returned interval is calculated using the following formula:

randomized interval = (interval * (random value in range [1 - factor, 1 + factor]))

next() will range between the randomization factor percentage below and above the retry interval.

For example, given the following parameters:

- interval = 0.2
- factor = 0.5

- multiplier = 2

the actual backoff period used in the next retry attempt will range between 1 and 3 seconds, multiplied by the exponential, that is, between 2 and 6 seconds.

Note: *max_internval* caps the *interval* and not the randomized interval.

If the time elapsed since an *ExponentialBackOff* instance is created goes past the *max_elapsed* time, then the method *next()* starts returning *Backoff.STOP*.

The elapsed time can be reset by calling *reset()*‘.

Example: Given the following default arguments, for 10 tries the sequence will be, and assuming we go over the *max_elapsed* on the 10th try:

```
Request #  RetryInterval (seconds)  Randomized Interval (seconds)
1          0.5                     [0.25,   0.75]
2          0.75                    [0.375,  1.125]
3          1.125                   [0.562,  1.687]
4          1.687                   [0.8435, 2.53]
5          2.53                    [1.265,  3.795]
6          3.795                   [1.897,  5.692]
7          5.692                   [2.846,  8.538]
8          8.538                   [4.269, 12.807]
9          12.807                  [6.403, 19.210]
10         19.210                  Backoff.STOP
```

For the opposite backoff strategy, see *riprova.ConstantBackoff*.

ExponentialBackOff is expected to run in a single-thread context.

Parameters

- **interval** (*int*) – interval time in seconds. Defaults to *500*.
- **factor** (*int | float*) – multiplier factor for exponential retries. Defaults to *0.5*. It should be between *0* and *1* number range.
- **max_interval** (*int*) – max allowed internval in seconds. Defaults to *60*.
- **max_elapsed** (*int*) – max elapsed total allowed time in seconds. Defaults to *15* minutes == *15 * 60* seconds.
- **multiplier** (*int | float*) – exponential multiplier. Defaults to *1.5*.

Raises AssertionError – in case of invalid params.

Usage:

```
@riprova.retry(backoff=riprova.ExponentialBackOff(interval=100))
def task(x):
    return x * x
```

elapsed

Returns the elapsed time since an *ExponentialBackOff* instance is created and is reset when *reset()* is called.

next()

Returns the number of seconds to wait before the next try, otherwise returns *Backoff.STOP*, which indicates the max number of retry operations were reached.

Returns time to wait in seconds before the next try.

Return type int

reset()
Reset the interval back to the initial retry interval and restarts the timer.

class riprova.**ErrorWhitelist**(*errors=None*)
Bases: object

Stores an error whitelist and provides a simple interface for whitelist update and mutation.

Parameters **errors** (*set|list|tuple[Exception]*) – optional list of error exceptions classes to whitelist.

errors
list of whilelist errors.

Type list

WHITELIST = set([<class 'riprova.exceptions.NotRetriableError'>, <type 'exceptions.Ref

add(**errors*)
Adds one or multiple error classes to the current whitelist.

Parameters ***errors** (*Exception*) – variadic error classes to add.

errors
Sets a new error whitelist, replacing the existent one.

Parameters **errors** (*list|tuple[Exception]*) – iterable containing errors to whitelist.

isretry(*error*)
Checks if a given error object is whitelisted or not.

Returns bool

class riprova.**ErrorBlacklist**(*errors=None*)
Bases: riprova.errors.ErrorWhitelist

Provides errors blacklist used to determine those exception errors who should be retried.

Implements the opposite behaviour to *ErrorWhitelist*.

Parameters **errors** (*set|list|tuple[Exception]*) – optional list of error exceptions classes to blacklist.

errors
list of blacklist errors.

Type list

isretry(*error*)
Checks if a given error object is not whitelisted.

Returns bool

riprova.**add_whitelist_error**(**errors*)
Add additional custom errors to the global whitelist.

Raises exceptions that are instances of the whitelisted errors won't be retried and they will be re-raised instead.

Parameters ***errors** (*Exception*) – variadic error classes to whitelist.

Usage:

```
riprova.add_whitelist_error(MyCustomError, AnotherError)
```

exception riprova.**RetryError**
Bases: exceptions.Exception

Retry error raised by this library will be an instance of this class.

Retry originated errors internally raised by *riprova* will be an instance of *RetryError* class.

exception riprova.**MaxRetriesExceeded**
Bases: riprova.exceptions.RetryError

Retry error raised when a maximum of retry attempts is exceeded, reported by the backoff strategy being used.

exception riprova.**RetryTimeoutError**
Bases: riprova.exceptions.RetryError

" Custom retry timeout error internally used by *riprova* when a task exceeds the maximum time execution limit.

exception riprova.**NotRetriableError**
Bases: exceptions.Exception

" Utility error class that can be inherited by developers for those cases where the error should be ignored by *riprova* retry engine.

__retry__
optional magic attribute inferred by *riprova* in order to determine when an error should be retried or not. You can optionally flag any error exception with this magic attribute in order to modify the retry behaviour. Defaults to *False*.

Type bool

Usage:

```
# Raised exception with the following ignored won't be retried
# by riprova
class MyWhiteListedError(riprova.NotRetriableError):
    pass

# You can optionally reverse that behaviour for specific cases
# by defining the `__retry__` magic attribute as `True`
class MyWhiteListedError(riprova.NotRetriableError):
    __retry__ = True
```

6.3 v0.3.0 / 2023-05-20

- Deprecate asyncio.corouting
- Drop Python 2 and 3.4 support

6.4 v0.2.7 / 2018-08-24

- Merge pull request #20 from ffix/forward-exception-instance
- Correct linter warnings
- Re-raise exception instance instead of new exception with no args
- Merge pull request #19 from EdwardBetts/spelling
- Correct spelling mistakes.

- feat(setup): support Python 3.7
- feat(History): add version changes

6.5 v0.2.6 / 2018-04-14

- fix(#17): handle as legit retriable error Timeout exceptions.

6.6 v0.2.5 / 2018-03-21

- Merge pull request #15 from jstasiak/allow-newer-six
- Allow newer six
- feat(History): update changes

6.7 v0.2.5 / 2018-03-21

- Merge pull request #15 from jstasiak/allow-newer-six
- Allow newer six
- feat(History): update changes

6.8 v0.2.4 / 2018-03-20

- merge(#14): Allow subsecond maxtimes for ExponentialBackoff

6.9 v0.2.3 / 2017-01-13

- refactor(retry): remove unnecessary partial function
- fix(retry): rename keyword param for partial application
- feat(docs): improve description
- refactor(Makefile): update publish task

6.10 v0.2.2 / 2017-01-06

- feat(package): add wheel distribution

6.11 v0.2.1 / 2017-01-04

- fix(retrier): remove debug print statement

6.12 v0.2.0 / 2017-01-02

- feat(core): use seconds as default time unit (introduces API breaking changes)
- refactor(examples): update examples to use new time unit
- feat(contextmanager): adds context manager support
- feat(examples): add context manager example
- feat: add context managers support

6.13 v0.1.3 / 2016-12-30

- refactor(async_retrier): simplify coroutine wrapper
- feat(whitelist): add whitelist and blacklist support
- feat(tests): add missing test cases for whitelist
- feat(retry): pass error_evaluator param
- fix(retrier): cast delay to float
- fix(tests): use valid exception for Python 2.7
- feat(#6): add custom error whilelist and custom error evaluator function
- Merge pull request #8 from tsarpaul/master
- refactor(decorator): do not expose retrier instance

6.14 v0.1.2 / 2016-12-27

- fix(decorator): wrap retries instance per function call

6.15 v0.1.1 / 2016-12-27

- fix(#2): handle and forward `asyncio.CancelledError` as non-retriable error

6.16 v0.1.0 / 2016-12-25

- First version

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index

Symbols

A

B

C

E

F

I

M

N

O

R

S

T

W