Backoff-Utils Documentation

Release 1.0.1

Insight Industry Inc.

Jul 11, 2020

Contents:

1 Getting Started **3**
- 1.1 Installation . . . 3
- 1.2 Hello, World . . . 4
- 1.3 Library Capabilities . . . 4

2 Using the Library **7**
- 2.1 Why are Backoff Strategies useful? . . . 8
- 2.2 How does this library help? . . . 8
- 2.3 Installing the Library . . . 8
- 2.4 Importing the Library . . . 8
- 2.5 Using the Backoff Function Call . . . 9
- 2.6 Using the Decorator Approach . . . 14
- 2.7 Stacking / Nesting / Chaining Strategies . . . 15

3 Strategies Explained **17**
- 3.1 Why Are Backoff Strategies Useful? . . . 17
- 3.2 How Do Strategies Work? . . . 18
- 3.3 Strategy Features . . . 18
- 3.4 Supported Strategies . . . 19
- 3.5 Creating Your Own Strategies . . . 21

4 API Reference **23**
- 4.1 `backoff()` Function . . . 23
- 4.2 `@apply_backoff()` Decorator . . . 25
- 4.3 Strategies . . . 27
- 4.4 Meta-classes . . . 34

5 Contributing to Backoff-Utils **37**
- 5.1 Design Philosophy . . . 38
- 5.2 Style Guide . . . 38
- 5.3 Dependencies . . . 41
- 5.4 Preparing Your Development Environment . . . 41
- 5.5 Ideas and Feature Requests . . . 41
- 5.6 Testing . . . 42
- 5.7 Submitting Pull Requests . . . 42
- 5.8 Building Documentation . . . 42
- 5.9 References . . . 42

6 Testing the Backoff-Utils **43**
6.1 Testing Philosophy . . . 43
6.2 Test Organization . . . 44
6.3 Configuring & Running Tests . . . 44
6.4 Skipping Tests . . . 45
6.5 Incremental Tests . . . 45

7 Release History **47**
7.1 Release 1.0.1 . . . 47
7.2 Release 1.0.0 . . . 47

8 Glossary **49**

9 Installation **51**
9.1 Importing . . . 51
9.2 Dependencies . . . 51

10 Hello, World Example **53**

11 Why Backoff-Utils? **55**

12 Library Features **57**
12.1 Supported Strategies . . . 57
12.2 Additional Features . . . 57

13 Feedback, Support, and Contributing **59**

14 Indices and tables **61**

Python Module Index **63**

Index **65**

Python Library for Backoff/Retry Strategies

Version Compatability

The **Backff-Utils** are designed to be compatible with Python 2.7 and Python 3.4 or higher.

Branch	Unit Tests
latest	
v. 1.0	
develop	

CHAPTER 1

Getting Started

- *Installation*
  - *Importing*
  - *Dependencies*
- *Hello, World*
- *Library Capabilities*

1.1 Installation

To install **Backoff-Utils**, just execute:

```
$ pip install backoff-utils
```

1.1.1 Importing

Once installed, to import **Backoff-Utils** into your project you can use:

```
#: Import the backoff() function.
from backoff_utils import backoff

#: Import the @apply_backoff() decorator.
from backoff_utils import apply_backoff

#: Import backoff strategies.
from backoff_utils import strategies
```

1.1.2 Dependencies

By design, **Backoff-Utils** are designed to rely on minimal dependencies. The only dependency they have outside of the Python standard library is:

- validator-collection which provides for robust validation functionality.

  This library in turn has two external dependencies: * jsonschema, and * (when installed under Python 2.7) regex

  which is a drop-in replacement for Python 2.7's (buggy) standard `re` module.

1.2 Hello, World

As a quick reference, here are some examples. Each of the examples below performs up to three attempts, applying an *exponential backoff* strategy with default configuration:

```
from backoff_utils import strategies

# Using a Function Call
from backoff_utils import backoff

def some_function(arg1, arg2, kwarg1 = None):
    # your code goes here
    pass

result = backoff(some_function,
                 args = ['value1', 'value2'],
                 kwargs = { 'kwarg1': 'value3' },
                 max_tries = 3,
                 max_delay = 3600,
                 strategy = strategies.Exponential)

# Using a Decorator
from backoff_utils import backoff

@apply_backoff(strategy = strategies.Exponential, max_tries = 3, max_delay = 3600)
def some_decorated_function(arg1, arg2, kwarg1 = None):
    # your code goes here
    pass

result = some_decorated_function('value1', 'value2', kwarg1 = 'value3')
```

1.3 Library Capabilities

There are two ways in which you can apply a backoff/retry strategy using the **Backoff-Utils**. Which approach you want to use will probably depend on your code and your code conventions:

- *using a function call*
- *using a decorator*

Both of these approaches support the following backoff strategies:

- *Exponential*

- *Fibonaccial*
- *Fixed*
- *Linear*
- *Polynomial*
- *custom strategies*

While the library's defaults are usable out-of-the-box, your backoff strategy can be further tailored to your needs. The library also supports:

- *random jitter*
- *argument-adjustment on retry*
- *selective exception capture*
- *chained backoff strategies*
- *failure handlers*
- *success handlers*
- *cut-off after a max delay*
- *cut-off after max tries*
- *scaling*
- *minimum delay*

See also:

While the **Backoff-Utils** are very straightforward to use, we recommend you review *Using the Library* to learn more about what it can do, and for a deep dive please see the *API Reference* .

CHAPTER 2

Using the Library

- *Why are Backoff Strategies useful?*
- *How does this library help?*
- *Installing the Library*
- *Importing the Library*
- *Using the Backoff Function Call*
  - *Basic Usage*
  - *Alternative Fallbacks*
  - *Retrying on Specific Errors*
  - *Handling Failures*
  - *Handling Success*
- *Using the Decorator Approach*
  - *Basic Usage*
  - *Alternative Fallbacks*
  - *Retrying on Specific Errors*
  - *Handling Failures*
  - *Handling Success*
- *Stacking / Nesting / Chaining Strategies*
  - *Using the Function Approach*
  - *Using the Decorator Approach*

2.1 Why are Backoff Strategies useful?

Because now and again, stuff breaks.

Often, when making function calls, something goes wrong. The internet might glitch. The API we're calling might timeout. Gremlins might eat your packets. Any number of things can go wrong, and Murphy's law tells us that they will.

Which is why we need *backoff strategies*. Basically, a backoff strategy is a technique that we can use to retry failing function calls after a given delay - and keep retrying them until either the function call works, or until we've tried so many times that we just give up and handle the error.

2.2 How does this library help?

This library provides a simple one-line approach to using backoff strategies in your code.

It provides a simple function (`backoff()`) and a simple decorator (`@apply_backoff()`) that let you easily retry problematic functions using five different configurable *strategies*.

2.3 Installing the Library

To install **Backoff-Utils**, just execute:

```
$ pip install backoff-utils
```

2.4 Importing the Library

There are three parts to the library that you should be aware of:

1. The `backoff()` function, which you can use to to apply a backoff strategy to a given function/method call inside your code.
2. The `@apply_backoff()` decorator, which you can use to always apply a backoff strategy to one of your function/methods.
3. The *Strategies Explained* module, which exposes the `BackoffStrategy` classes that you supply to the function and decorator, telling them how to delay between attempts. The specific strategies are:
    - `strategies.Exponential`
    - `strategies.Fibonacci`
    - `strategies.Fixed`
    - `strategies.Linear`
    - `strategies.Polynomial`

All three of these components are importable directly from the `backoff_utils` package as shown below:

```
#: Import everything
from backoff_utils import backoff, apply_backoff, strategies

#: Import the backoff() function.
from backoff_utils import backoff

#: Import the @apply_backoff() decorator.
from backoff_utils import apply_backoff

#: Import backoff strategies.
from backoff_utils import strategies
```

2.5 Using the Backoff Function Call

You use the *backoff()* function when:

- you want to call some other function/method using a backoff strategy, but that function/method is not decorated with `@apply_backoff()`
- you want to call some other function/method using a backoff strategy, but if that call fails, you want to retry using a different call.

Tip: The function approach is often used when we want to apply a backoff strategy to a function or method called in someone else's code, like in some imported third-party library.

Since that code won't be using the *@apply_backoff()* decorator, if we want to apply a backoff strategy we'll need to use the *backoff()* function.

2.5.1 Basic Usage

For example, let's imagine we have a function:

```
def some_function(arg1, arg2, kwarg1 = value):
    # Function does stuff here
```

When our code calls `some_function()`, we want to apply an *Exponential* backoff strategy. We can do so using:

```
result = backoff(some_function,
                 args = ['value1', 'value2'],
                 kwargs = { 'kwarg1': 'value3' },
                 max_tries = 3,
                 max_delay = 30,
                 strategy = strategies.Exponential)
```

Let's breakdown what this does. First, it will try calling:

```
result = some_function('value1', 'value2', kwarg1 = 'value3')
```

If this raises an error, it will retry using an *Exponential* delay. It will continue to retry, until either it has made 3 attempts or 30 seconds have elapsed. If this call is still failing after 3 attempts or 30 seconds, it will raise the last `Exception` raised by `some_function()`.

Note: The `strategy` argument can accept either a class that inherits from *BackoffStrategy*, or it can accept an *instance* of a class that inherits from *BackoffStrategy*.

Passing a class will use the default configuration for the backoff strategy, while passing an instance will let you modify that configuration. For example:

```
my_strategy = strategies.Polynomial(exponent = 3, scale_factor = 0.5)

result = backoff(some_function,
                 args = ['value1', 'value2'],
                 kwargs = { 'kwarg1': 'value3' },
                 max_tries = 3,
                 max_delay = 30,
                 strategy = my_strategy)
```

will call `some_function()` with a *Polynomial* strategy using an exponent of 3 and a *scale factor* of 0.5.

See also:

For more information, please see: *Strategies Explained*.

Tip: If you don't supply a `max_tries` argument, the backoff strategy will look for a default max in the `BACKOFF_DEFAULT_TRIES` environment variable. If that environment variable doesn't exist, it will retry your call three times then fail.

If you don't supply a `max_delay`, the backoff strategy look for a default maximum delay in the `BACKOFF_DEFAULT_DELAY` environment variable. If that environment variable doesn't exist, it will keep retrying your call until it hits `max_tries`.

And that's it!

See also:

For more detailed documentation, please see the *API Reference* for the *backoff()* function.

2.5.2 Alternative Fallbacks

The *backoff()* function allows you to fallback to either a different function or a different set of arguments after the first failure.

For example, let's imagine a situation where we have two functions:

```
def some_function(arg1, arg2, kwarg1 = None):
    # Function does stuff.

def some_alternative_function(arg1, arg2, arg3, arg4):
    # Function does stuff.
```

Now, let's try to first call `some_function()`, and if that doesn't work, we can automatically try calling `some_alternative_function()` after our delay:

```
result = backoff(some_function,
                 args = ['value1', 'value2'],
                 kwargs = { 'kwarg1': 'value3' },
                 retry_execute = some_alternative_function,
                 retry_args = ['value1', 'value2', 'value3', 'something else'],
                 retry_kwargs = {},
                 max_tries = 3,
                 max_delay = 30,
                 strategy = strategies.Exponential)
```

Let's breakdown what this will do. As before, first it will try calling:

```
result = some_function('value1', 'value2', kwarg1 = 'value3')
```

When that doesn't work, it will then try calling:

```
result = some_alternative_function('value1' ,'value2', 'value3', 'something else')
```

until either that is successful, or the strategy exceeds the maximum number of tries or the maximum delay. If everything fails, then it will raise the last `Exception` raised by `some_alternative_function()`.

2.5.3 Retrying on Specific Errors

Not all errors are created equal. For some errors, we know with 100% certainty that retrying a function/method call with the same parameters will produce the exact same error every time. Which means there's no point to applying a backoff strategy. However, certain errors may be caused by other factors... which means that if we try again, the function/method call might just work.

This is often the cause when a function/method is making a call across a network (like an HTTP request). Such a request might timeout because the API just happened to be over-burdened when the first request was made. If you want a second, maybe your next request will get through.

The *`backoff()`* function allows you to only apply the backoff strategy for a defined set of exceptions. If the function/method you're trying raises an exception that isn't on the list? Then the call won't be retried.

Let's assume we have `some_function()` as follows:

```
def some_function(arg1, arg2, kwarg1 = None):
    # Function does stuff.
```

Now, let's further assume that `some_function()` will sometimes raise:

- `TimeoutError`
- `IOError`
- `NotImplementedError`

If we get `NotImplementedError`, there's no point in retrying: The same arguments will always produce the same error. But the other two errors may just be a momentary glitch, and retrying after some delay may work. Here's how we would do that:

```
result = backoff(some_function,
                 args = ['value1', 'value2'],
                 kwargs = { 'kwarg1': 'value3' },
                 max_tries = 3,
                 max_delay = 30,
```

(continues on next page)

(continued from previous page)

```
                catch_exceptions = [type(TimeoutError), type(IOError)],
                strategy = strategies.Exponential)
```

Now, when `some_function('value1', 'value2', kwarg1 = 'value3')` raises a `TimeoutError` or `IOError`, the call will be retried up to 3 times or for 30 seconds (whichever comes first). If the call raises any other exception, then the call will fail and bubble that exception up to your code where you'll need to handle it.

Caution: If `catch_exceptions` is not `None` (the default, which will catch all exceptions), then it is very important that the `catch_exceptions` argument always contain one or more `type(Exception())` values. For example:

```
# GOOD: This will work.
result = backoff(some_function,
                 args = ['value1', 'value2'],
                 kwargs = { 'kwarg1': 'value3' },
                 max_tries = 3,
                 max_delay = 30,
                 catch_exceptions = [type(TimeoutError()), type(IOError())],
                 strategy = strategies.Exponential)

result = backoff(some_function,
                 args = ['value1', 'value2'],
                 kwargs = { 'kwarg1': 'value3' },
                 max_tries = 3,
                 max_delay = 30,
                 catch_exceptions = type(TimeoutError()),
                 strategy = strategies.Exponential)

# BAD: This will not work.
result = backoff(some_function,
                 args = ['value1', 'value2'],
                 kwargs = { 'kwarg1': 'value3' },
                 max_tries = 3,
                 max_delay = 30,
                 catch_exceptions = [TimeoutError, IOError],
                 strategy = strategies.Exponential)

result = backoff(some_function,
                 args = ['value1', 'value2'],
                 kwargs = { 'kwarg1': 'value3' },
                 max_tries = 3,
                 max_delay = 30,
                 catch_exceptions = [type(TimeoutError), type(IOError)],
                 strategy = strategies.Exponential)
```

2.5.4 Handling Failures

Sometimes, even after retrying stuff, your function/method call will still fail. That's life. But when that happens, you might want to call some *other* function or method to do something in response. You can do this by passing that function/method to the *`backoff()`* function as the `on_failure` argument.

For example, let's imagine we have two functions:

```
def some_function(arg1, arg2, kwarg1 = None):
    # Function does stuff.

def error_handler(*args, **kwargs):
    # Function does stuff.
```

We can have the backoff strategy call `error_handler()` when it has a final failure - meaning after *`backoff()`* has tried and failed multiple times, after it has timed out, or if `some_function()` raises an exception that is not listed in `catch_exceptions`.

Here's how that would look:

```
result = backoff(some_function,
                 args = ['value1', 'value2'],
                 kwargs = { 'kwarg1': 'value3' },
                 max_tries = 3,
                 max_delay = 30,
                 catch_exceptions = [type(TimeoutError()), type(IOError())],
                 on_failure = error_handler,
                 strategy = strategies.Exponential)
```

Tip: If you pass a class that descends from `Exception` to `on_failure`, that exception will be raised with the message of the last exception raised by `some_function()`.

Caution: If you are passing a custom function (*not* an `Exception`) to `on_failure`, that custom function must accept three positional arguments:

1. `error` - the last exception raised
2. `message` - the message of the last exception raised
3. `traceback` - the stack trace associated with the last exception raised

If the `on_failure` function cannot accept those three positional arguments, or if the `on_failure` function itself fails, then the last exception raised will bubble up.

2.5.5 Handling Success

So we've talked a lot about failures here. But sometimes, things work! When the *`backoff()`* function is successful, it will always return the value back to where it was called. But sometimes, you want to fire a success handler before that value is returned. You can do this by passing a handler function to the *`backoff()`* function's `on_success` argument.

Let's imagine we have the following:

```
def some_function(arg1, arg2, kwarg1 = None):
    # Function does stuff.

def success_handler(value_on_success):
    # Function does stuff.

result = backoff(some_function,
                 args = ['value1', 'value2'],
```

(continues on next page)

(continued from previous page)

```
                kwargs = { 'kwarg1': 'value3' },
                max_tries = 3,
                max_delay = 30,
                catch_exceptions = [type(TimeoutError()), type(IOError())],
                on_success = success_handler,
                strategy = strategies.Exponential)

# some more stuff happens here
```

Now, when `some_function()` is successful, *before* `result` is returned to your code, the *`backoff()`* function will call:

```
success_handler(result)
```

When `success_handler()` returns control, the *`backoff()`* function will return `result` and your code can continue.

Caution: It is very important that your `on_success` function always accept a single `result` value. This will always be the value returned by function/method you were trying to call using a backoff strategy.

Tip: A common pattern is to make your `on_success` function an asynchronous function. This can help parallelize your code to some extent, which means your code isn't waiting for your `on_success` handler to complete before continuing.

2.6 Using the Decorator Approach

You use the *`@apply_backoff()`* decorator when you want to *always* apply a particular backoff strategy to one of your functions or methods.

2.6.1 Basic Usage

For example, let's imagine we have a function:

```
def some_function(arg1, arg2, kwarg1 = value):
    # Function does stuff here

result = some_function('value1', 'value2', kwarg1 = 'value3')
```

Whenever your code calls `some_function()`, we want to apply an *`Exponential`* backoff strategy for a maximum of 5 tries provided they don't take longer than 30 seconds. Here's how we would do that:

```
@apply_backoff(strategies.Exponential, max_tries = 5, max_delay = 30)
def some_function(arg1, arg2, kwarg1 = value):
    # Function does stuff here

result = some_function('value1', 'value2', kwarg1 = 'value3')
```

That's it! Now, whenever you call `some_function()`, the decorator will look for an error, and if it catches one, will retry the call after an exponential delay. It will keep retrying until it has tried five times, or until 30 seconds have passed - whichever is first.

Note: Just as when using the *function call approach*, you can pass the `BackoffStrategy`, the number of `max_tries`, and `max_delay` to the `@apply_backoff` decorator.

See also:

For more detailed documentation, please see the *API Reference* for the `@apply_backoff()` decorator.

2.6.2 Alternative Fallbacks

Caution: The `@apply_backoff()` decorator does not support alternative fallbacks. If you want to use alternative fallbacks, then we suggest using the *function approach*.

2.6.3 Retrying on Specific Errors

When using the `@apply_backoff()` decorator, you can retry on specific errors by passing those error types to the decorator's `catch_exceptions` argument.

See also:

This works the same as the `catch_exceptions` argument when using the *function call approach*.

2.6.4 Handling Failures

See also:

When using the `@apply_backoff()` decorator, you can fire an `on_failure` handler by passing an `on_failure` argument just as you can for the *function call approach*.

2.6.5 Handling Success

See also:

When using the `@apply_backoff()` decorator, you can fire an `on_success` handler by passing an `on_success` argument just as you can for the *function call approach*.

2.7 Stacking / Nesting / Chaining Strategies

Let's imagine that the function/method you want to call will raise two different errors, and you want to apply a *different* backoff strategy for each error. Using the library, that's fairly straightforward.

For example, let's imagine we have a function:

```
def some_function(arg1, arg2, kwarg1 = None):
    # Function does stuff.
```

which sometimes raises a `TimeoutError` and sometimes an `IOError`.

Let's further assume that if it raises a `TimeoutError`, we want to apply an `Exponential` strategy up to five times, but for an `IOError` we want to apply a `Linear` strategy up to 3 times.

2.7.1 Using the Function Approach

Here's how we could do that using the function approach:

```
def backoff_for_timeout():
    return backoff(some_function,
                   args = ['value1', 'value2'],
                   kwargs = { 'kwarg1': 'value3' },
                   max_tries = 5,
                   catch_exceptions = [type(TimeoutError())],
                   strategy = strategies.Exponential)

result = backoff(backoff_for_timeout,
                 max_tries = 3,
                 catch_exceptions = [type(IOError())],
                 strategy = strategies.Linear)
```

First, your code will call the *`backoff()`* function for `backoff_for_timeout()`. It will be looking to catch any `IOError` that `backoff_for_timeout()` raises. When it catches one, it will retry up to three times using the *`Linear`* strategy.

When the *`backoff()`* function calls `backoff_for_timeout()`, that function will in turn call another *`backoff()`* function for `some_function()`. It will be looking to catch any `TimeoutError` exceptions that `some_function()` raises. When it catches one, it will retry up to five times using the *`Exponential`* strategy.

At this point, if `some_function()` raises an `IOError`, however, it will bubble up to the first *`backoff()`* function, which will catch and handle it.

2.7.2 Using the Decorator Approach

Here's how we could do it using the *`@apply_backoff`* decorator:

```
@apply_backoff(strategies.Linear, max_tries = 3, catch_exceptions = type(IOError))
@apply_backoff(strategies.Exponential, max_tries = 5, catch_exceptions =␣
↪type(TimeoutError))
def some_function(arg1, arg2, kwarg1 = None):
    # Function does stuff.

result = some_function('value1', 'value2', kwarg1 = 'value3')
```

Now, when your code calls `some_function()`, it will first try to catch any `TimeoutError` raised by `some_function()`. If it catches one, it will retry `some_function()` up to 5 times using an *`Exponential`* strategy.

If `some_function()` raises anything other than a `TimeoutError`, that error will bubble up to the *next* decorator you've applied. That decorator looks for a `IOError`. If it catches one, it will retry up to 3 times using a *`Linear`* strategy.

CHAPTER 3

Strategies Explained

- *Why Are Backoff Strategies Useful?*
- *How Do Strategies Work?*
- *Strategy Features*
  - *Random Jitter*
  - *Minimum Delay*
  - *Scale Factor*
- *Supported Strategies*
  - *Exponential*
  - *Fibonacci*
  - *Fixed*
  - *Linear*
  - *Polynomial*
- *Creating Your Own Strategies*

3.1 Why Are Backoff Strategies Useful?

Because now and again, stuff breaks.

Often, when making function calls, something goes wrong. The internet might glitch. The API we're calling might timeout. Gremlins might eat your packets. Any number of things can go wrong, and Murphy's law tells us that they will.

Which is why we need *backoff strategies*. Basically, a backoff strategy is a technique that we can use to retry failing function calls after a given delay - and keep retrying them until either the function call works, or until we've tried so many times that we just give up and handle the error.

3.2 How Do Strategies Work?

In the **Backoff-Utils** library, strategies exist to calculate the delay that should be applied between retries. That's all they do. Everything else is handled by the *backoff()* function and *@apply_backoff* decorator.

The library supports five different strategies, each of which inherits from *BackoffStrategy*.

> **Caution:** *BackoffStrategy* is itself an abstract base class and cannot be instantiated directly. You can subclass it to create your own custom strategies, or you can supply one of our ready-made strategies as the `strategy` argument when applying a backoff.

When you apply a backoff strategy, you must supply a `strategy` argument which can accept either a class that inherits from *BackoffStrategy*, or an *instance* of a class that inherits from *BackoffStrategy*.

Passing a class will use the default configuration for the backoff strategy, while passing an instance will let you modify that configuration. For example:

```
result = backoff(some_function,
                 args = ['value1', 'value2'],
                 kwargs = { 'kwarg1': 'value3' },
                 max_tries = 3,
                 max_delay = 30,
                 strategy = strategies.Exponential)
```

will call `some_function()` with an *Exponential* strategy applying its default settings, while:

```
my_strategy = strategies.Polynomial(exponent = 3, scale_factor = 0.5)

result = backoff(some_function,
                 args = ['value1', 'value2'],
                 kwargs = { 'kwarg1': 'value3' },
                 max_tries = 3,
                 max_delay = 30,
                 strategy = my_strategy)
```

will call `some_function()` with a *Polynomial* strategy using an exponent of 3 and a *scale factor* of 0.5.

3.3 Strategy Features

3.3.1 Random Jitter

All strategies support using a random *jitter*.

You can deactivate the jitter on a strategy by instantiating it with the argument `jitter = False`. For example:

```
my_strategy = strategies.Exponential(jitter = False)
```

will ensure that no jitter is applied.

Hint: By default, all strategies apply a random *jitter* unless explicitly deactivated.

3.3.2 Minimum Delay

While each strategy calculates its delay based on its own logic, you can ensure that the delay returned is always a certain minimum number of seconds. You can apply a minimum by instantiating a strategy with the `minimum` argument. For example:

```
my_strategy = strategies.Exponential(minimum = 5)
```

will ensure that at least 5 seconds will pass between retry attempts.

Hint: By default, there is no minimum.

3.3.3 Scale Factor

Certain strategies - like the *`Polynomial`* strategy - can rapidly lead to very long delays between retry attempts. To offset this, while still retaining the shape of the curve between retry attempts, each strategy has a `scale_factor` property which is multipled by the "unadjusted" delay. This can be used to reduce (or increase) the size (technically the magnitude) of the delay.

To apply a *scale factor*, pass it as the `scale_factor` argument when instantiating the strategy. For example:

```
my_strategy = strategies.Exponential(scale_factor = 0.5)
```

will ensure that whatever delay is calculated will always be reduced by 50% before being applied.

Hint: The *scale factor* defaults to a value of `1.0`.

3.4 Supported Strategies

The library comes with five commonly-used backoff/retry strategies:

- *Exponential*
- *Fibonaccial*
- *Fixed*
- *Linear*
- *Polynomial*

However, you can also create your own *custom strategies* by inheriting from *`BackoffStrategy`*.

3.4.1 Exponential

The base delay time is calculated as:

$$2^a$$

where a is the number of unsuccessful attempts that have been made.

3.4.2 Fibonacci

The base delay time is returned as the Fibonacci number corresponding to the current attempt.

3.4.3 Fixed

The base delay time is calculated as a fixed value determined by the attempt number.

To configure the sequence, instantiate the strategy passing an iterable to `sequence` like in the example below:

```
my_strategy = strategies.Fixed(sequence = [2, 4, 6, 8])
```

Note: If the number of attempts exceeds the length of the sequence, the last delay in the sequence will be repeated.

Tip: If no sequence is given, by default each base delay will be 1 second long.

3.4.4 Linear

The base delay time is equal to the attempt count.

3.4.5 Polynomial

The base delay time is calculated as:

$$a^e$$

where:

- a is the number of unsuccessful attempts that have been made,
- e is the `exponent` configured for the strategy.

To set the exponent, pass `exponent` as an argument to the class as follows:

```
my_strategy = strategies.Polynomial(exponent = 2)
```

will calculate the base delay as

$$a^2$$

where a is the number of unsuccessful attempts that have been made.

3.5 Creating Your Own Strategies

You can create your own custom backoff strategy by subclassing from *strategies.BackoffStrategy*. When you do so, you will need to define your own `time_to_sleep` property which returns a `float`.

For example:

```
import random
from backoff_utils import strategies

class MyCustomStrategy(strategies.BackoffStrategy):
  """This is a custom strategy that will always wait a random number of
  milliseconds."""

  @property
  def time_to_sleep(self):
    return random.random()
```

The custom strategy created above will always wait a random number of milliseconds, regardless of anything else. You can make your classes as complicated as they need to be, and use whatever logic you choose.

CHAPTER 4

API Reference

- `backoff()` *Function*
- `@apply_backoff()` *Decorator*
- *Strategies*
  - *Exponential*
  - *Fibonacci*
  - *Fixed*
  - *Linear*
  - *Polynomial*
- *Meta-classes*
  - *BackoffStrategy*

4.1 `backoff()` Function

backoff(*to_execute*, *args=None*, *kwargs=None*, *strategy=None*, *retry_execute=None*, *retry_args=None*, *retry_kwargs=None*, *max_tries=None*, *max_delay=None*, *catch_exceptions=None*, *on_failure=None*, *on_success=None*)

Retry a function call multiple times with a delay per the strategy given.

Parameters

- **to_execute** (*callable*) – The function call that is to be attempted.
- **args** (iterable / None.) – The positional arguments to pass to the function on the first attempt.

If `retry_args` is `None`, will re-use these arguments on retry attempts as well.

- **kwargs** (`dict` / `None`) – The keyword arguments to pass to the function on the first attempt.

  If `retry_kwargs` is `None`, will re-use these keyword arguments on retry attempts as well.

- **strategy** (`BackoffStrategy`) – The `BackoffStrategy` to use when determining the delay between retry attempts.

  If `None`, defaults to `Exponential`.

- **retry_execute** (callable / `None`) – The function to call on retry attempts.

  If `None`, will retry `to_execute`.

  Defaults to `None`.

- **retry_args** (iterable / `None`) – The positional arguments to pass to the function on retry attempts.

  If `None`, will re-use `args`.

  Defaults to `None`.

- **retry_kwargs** – The keyword arguments to pass to the function on retry attempts.

  If `None`, will re-use `kwargs`.

  Defaults to `None`.

- **max_tries** (int / `None`) – The maximum number of times to attempt the call.

  If `None`, will apply an environment variable `BACKOFF_DEFAULT_TRIES`. If that environment variable is not set, will apply a default of `3`.

- **max_delay** (`None` / int) – The maximum number of seconds to wait befor giving up once and for all. If `None`, will apply an environment variable `BACKOFF_DEFAULT_DELAY` if that environment variable is set. If it is not set, will not apply a max delay at all.

- **catch_exceptions** (iterable of form `[type(exception()), ...]`) – The `type(exception)` to catch and retry. If `None`, will catch all exceptions.

  Defaults to `None`.

  Caution: The iterable must contain one or more types of exception *instances*, and not class objects. For example:

  ```
  # GOOD:
  catch_exceptions = (type(ValueError()), type(TypeError()))

  # BAD:
  catch_exceptions = (type(ValueError), type(ValueError))

  # BAD:
  catch_exceptions = (ValueError, TypeError)

  # BAD:
  catch_exceptions = (ValueError(), TypeError())
  ```

- **on_failure** (`Exception` / function / `None`) – The `exception` or function to call when all retry attempts have failed.

If None, will raise the last-caught exception.

If an exception, will raise the exception with the same message as the last-caught exception.

If a function, will call the function and pass the last-raised exception, its message, and stacktrace to the function.

Defaults to None.

- **on_success** (callable / None) – The function to call when the operation was successful. The function receives the result of the to_execute or retry_execute function that was successful, and is called before that result is returned to whatever code called the backoff function. If None, will just return the result of to_execute or retry_execute without calling a handler.

  Defaults to None.

Returns The result of the attempted function.

Example:

```
from backoff_utils import backoff

def some_function(arg1, arg2, kwarg1 = None):
    # Function does something
    pass

result = backoff(some_function,
                 args = ['value1', 'value2'],
                 kwargs = { 'kwarg1': 'value3' },
                 max_tries = 3,
                 max_delay = 30,
                 strategy = strategies.Exponential)
```

4.2 @apply_backoff() Decorator

apply_backoff(*strategy=None*, *max_tries=None*, *max_delay=None*, *catch_exceptions=None*, *on_failure=None*, *on_success=None*)

Decorator that applies a backoff strategy to a decorated function/method.

Parameters

- **strategy** (BackoffStrategy) – The BackoffStrategy to use when determining the delay between retry attempts. If None, defaults to Exponential.
- **max_tries** (int / None) – The maximum number of times to attempt the call.

  If None, will apply an environment variable BACKOFF_DEFAULT_TRIES. If that environment variable is not set, will apply a default of 3.

- **max_delay** (None / class:*int* <*python:int*>) – The maximum number of seconds to wait befor giving up once and for all.

  If None, will apply an environment variable BACKOFF_DEFAULT_DELAY if that environment variable is set. If it is not set, will not apply a max delay at all.

- **catch_exceptions** (iterable of form `[type(exception()), ...]`) – The `type(exception)` to catch and retry. If `None`, will catch all exceptions.

  Defaults to `None`.

  Caution: The iterable must contain one or more types of exception *instances*, and not class objects. For example:

  ```
  # GOOD:
  catch_exceptions = (type(ValueError()), type(TypeError()))

  # BAD:
  catch_exceptions = (type(ValueError), type(ValueError))

  # BAD:
  catch_exceptions = (ValueError, TypeError)

  # BAD:
  catch_exceptions = (ValueError(), TypeError())
  ```

- **on_failure** (`Exception` / function / `None`) – The `exception` or function to call when all retry attempts have failed.

  If `None`, will raise the last-caught `Exception`.

  If an `Exception`, will raise the exception with the same message as the last-caught exception.

  If a function, will call the function and pass the last-raised exception, its message, and stacktrace to the function.

  Defaults to `None`.

- **on_success** (callable / `None`) – The function to call when the operation was successful.

  The function receives the result of the decorated function, and is called before that result is returned to whatever code called the decorated function.

  If `None`, will just return the result of the decorated function without calling a handler.

  Defaults to `None`.

Example:

```
@apply_backoff(strategy = strategies.ExponentialBackoff,
               max_tries = 5,
               max_delay = 30)
def some_function(arg1, arg2, kwarg1 = None):
    pass

result = some_function('value1', 'value2', kwarg1 = 'value3')
```

4.3 Strategies

4.3.1 Exponential

class Exponential(*attempt=None*, *minimum=0.0*, *jitter=True*, *scale_factor=1.0*, ***kwargs*)

Implements the *exponential backoff* strategy.

The base delay time is calculated as:

$$2^a$$

where a is the number of the current attempt being made.

Parameters

- **attempt** (int) – The number of the attempt that was last-attempted. This value is used by the strategy to determine the amount of time to delay before continuing.
- **minimum** (*number*) – The minimum delay to apply. Defaults to `0`.
- **jitter** (bool) – If `True`, will add a random float to the delay. Defaults to `True`.
- **scale_factor** (float) – A factor by which the `time_to_sleep` is multiplied to adjust its scale. Defaults to `1.0`.

Class Attributes

IS_INSTANTIATED = False

Indicates whether the object is an instance of the strategy, or merely its class object.

Return type bool

Properties

attempt = None

The number of the attempt that the strategy is currently evaluating.

Return type int / None

minimum = 0.0

The minimum delay to apply, expressed in seconds.

Return type float

jitter = True

If `True`, will add a random float between 0 and 1 to the delay.

Return type bool

scale_factor = 1.0

A factor by which the `time_to_sleep` is multiplied to adjust its scale.

Return type float

time_to_sleep = 0.0 (read-only)

The base number of seconds to delay before allowing a retry.

Return type float

Class Methods

classmethod Exponential.**delay**(*attempt*, *minimum=None*, *jitter=None*, *scale_factor=1.0*)
Delay for a set period of time based on the `attempt`.

Parameters

- **attempt** (`int`) – The number of the attempt that was last-attempted. This value is used by the strategy to determine the amount of time to delay before continuing.
- **minimum** (*number*) – The minimum number of seconds to delay.

  If `None`, will apply either the strategy's default or the instance's configured property.
- **jitter** (`bool`) – If `True`, will add a random float to the delay.

  If `False`, will not.

  If `None`, will apply either the strategy's default or the instance's configured property.
- **scale_factor** (`float`) – A factor by which the `time_to_sleep` is multiplied to adjust its scale.

  If `None`, will apply either the strategy's default or the instance's configured property.

4.3.2 Fibonacci

class Fibonacci(*attempt=None*, *minimum=0.0*, *jitter=True*, *scale_factor=1.0*, ***kwargs*)
Implements the *fibonacci backoff* strategy.

The base delay time is returned as the Fibonacci number corresponding to the current attempt.

Parameters

- **attempt** (`int`) – The number of the attempt that was last-attempted. This value is used by the strategy to determine the amount of time to delay before continuing.
- **minimum** (*number*) – The minimum delay to apply. Defaults to `0`.
- **jitter** (`bool`) – If `True`, will add a random float to the delay. Defaults to `True`.
- **scale_factor** (`float`) – A factor by which the `time_to_sleep` is multiplied to adjust its scale. Defaults to `1.0`.

Class Attributes

IS_INSTANTIATED = False
Indicates whether the object is an instance of the strategy, or merely its class object.

Return type `bool`

Properties

attempt = None
The number of the attempt that the strategy is currently evaluating.

Return type `int` / `None`

minimum = 0.0
The minimum delay to apply, expressed in seconds.

Return type float

jitter = True
If True, will add a random float between 0 and 1 to the delay.

Return type bool

scale_factor = 1.0
A factor by which the time_to_sleep is multiplied to adjust its scale.

Return type float

time_to_sleep = 0.0 (read-only)
The base number of seconds to delay before allowing a retry.

Return type float

Class Methods

classmethod Fibonacci.**delay**(*attempt*, *minimum=None*, *jitter=None*, *scale_factor=1.0*)
Delay for a set period of time based on the attempt.

Parameters

- **attempt** (int) – The number of the attempt that was last-attempted. This value is used by the strategy to determine the amount of time to delay before continuing.
- **minimum** (*number*) – The minimum number of seconds to delay.

  If None, will apply either the strategy's default or the instance's configured property.
- **jitter** (bool) – If True, will add a random float to the delay.

  If False, will not.

  If None, will apply either the strategy's default or the instance's configured property.
- **scale_factor** (float) – A factor by which the time_to_sleep is multiplied to adjust its scale.

  If None, will apply either the strategy's default or the instance's configured property.

4.3.3 Fixed

class Fixed(*attempt=None*, *sequence=None*, *minimum=0*, *jitter=True*, *scale_factor=1.0*, ***kwargs*)
Implements the *fixed backoff* strategy.

The base delay time is calculated as a fixed value determined by the attempt number.

Parameters

- **attempt** (int) – The number of the attempt that was last-attempted. This value is used by the strategy to determine the amount of time to delay before continuing.
- **sequence** (*iterable of numbers*) – The sequence of base delay times to return based on the attempt number.
- **minimum** (*number*) – The minimum delay to apply. Defaults to 0.

- **jitter** (bool) – If True, will add a random float to the delay. Defaults to True.
- **scale_factor** (float) – A factor by which the time_to_sleep is multiplied to adjust its scale.

  Defaults to 1.0.

Class Attributes

IS_INSTANTIATED = False
Indicates whether the object is an instance of the strategy, or merely its class object.

Return type bool

Properties

attempt = None
The number of the attempt that the strategy is currently evaluating.

Return type int / None

minimum = 0.0
The minimum delay to apply, expressed in seconds.

Return type float

jitter = True
If True, will add a random float between 0 and 1 to the delay.

Return type bool

scale_factor = 1.0
A factor by which the time_to_sleep is multiplied to adjust its scale.

Return type float

sequence = None
The sequence of base delay times (in seconds) to return based on the attempt number.

Note: If the number of attempts exceeds the length of the sequence, the last delay in the sequence will be repeated.

Tip: If no sequence is given, by default each base delay will be 1 second long.

Return type iterable of numbers / None

time_to_sleep = 0.0 (read-only)
The base number of seconds to delay before allowing a retry.

Return type float

Class Methods

classmethod `Fixed.`**delay**(*attempt*, *minimum=None*, *jitter=None*, *scale_factor=1.0*)
Delay for a set period of time based on the `attempt`.

Parameters

- **attempt** (`int`) – The number of the attempt that was last-attempted. This value is used by the strategy to determine the amount of time to delay before continuing.
- **minimum** (*number*) – The minimum number of seconds to delay.

  If `None`, will apply either the strategy's default or the instance's configured property.
- **jitter** (`bool`) – If `True`, will add a random float to the delay.

  If `False`, will not.

  If `None`, will apply either the strategy's default or the instance's configured property.
- **scale_factor** (`float`) – A factor by which the `time_to_sleep` is multiplied to adjust its scale.

  If `None`, will apply either the strategy's default or the instance's configured property.

4.3.4 Linear

class Linear(*attempt=None*, *minimum=0.0*, *jitter=True*, *scale_factor=1.0*, ***kwargs*)
Implements the *fixed backoff* strategy.

The base delay time is equal to the attempt count.

Parameters

- **attempt** (`int`) – The number of the attempt that was last-attempted. This value is used by the strategy to determine the amount of time to delay before continuing.
- **minimum** (*number*) – The minimum delay to apply. Defaults to `0`.
- **jitter** (`bool`) – If `True`, will add a random float to the delay. Defaults to `True`.
- **scale_factor** (`float`) – A factor by which the `time_to_sleep` is multiplied to adjust its scale. Defaults to `1.0`.

Class Attributes

IS_INSTANTIATED = False
Indicates whether the object is an instance of the strategy, or merely its class object.

Return type `bool`

Properties

attempt = None
The number of the attempt that the strategy is currently evaluating.

Return type `int` / `None`

minimum = 0.0
The minimum delay to apply, expressed in seconds.

Return type float

jitter = True
If True, will add a random float between 0 and 1 to the delay.

Return type bool

scale_factor = 1.0
A factor by which the time_to_sleep is multiplied to adjust its scale.

Return type float

time_to_sleep = 0.0 (read-only)
The base number of seconds to delay before allowing a retry.

Return type float

Class Methods

classmethod Linear.**delay**(*attempt*, *minimum=None*, *jitter=None*, *scale_factor=1.0*)
Delay for a set period of time based on the attempt.

Parameters

- **attempt** (int) – The number of the attempt that was last-attempted. This value is used by the strategy to determine the amount of time to delay before continuing.
- **minimum** (*number*) – The minimum number of seconds to delay.

  If None, will apply either the strategy's default or the instance's configured property.
- **jitter** (bool) – If True, will add a random float to the delay.

  If False, will not.

  If None, will apply either the strategy's default or the instance's configured property.
- **scale_factor** (float) – A factor by which the time_to_sleep is multiplied to adjust its scale.

  If None, will apply either the strategy's default or the instance's configured property.

4.3.5 Polynomial

class Polynomial(*attempt=None*, *exponent=1*, *minimum=0*, *jitter=True*, *scale_factor=1.0*, ***kwargs*)
Implements the *polynomial backoff* strategy.

The base delay time is calculated as:

$$a^e$$

where:

- a is the number of attempts made
- e is the *exponent* property

Parameters

- **attempt** (float) – The number of the attempt that was last-attempted. This value is used by the strategy to determine the amount of time to delay before continuing.
- **exponent** (int) – The exponent to apply when calculating the base delay. Defaults to 1.
- **minimum** (*number*) – The minimum delay to apply. Defaults to 0.
- **jitter** (bool) – If True, will add a random float to the delay. Defaults to True.
- **scale_factor** (float) – A factor by which the time_to_sleep is multiplied to adjust its scale. Defaults to 1.0.

Class Attributes

IS_INSTANTIATED = False
Indicates whether the object is an instance of the strategy, or merely its class object.

Return type bool

Properties

attempt = None
The number of the attempt that the strategy is currently evaluating.

Return type int / None

exponent = 1.0
The exponent to apply when calculating the base delay. Defaults to 1.0.

Return type float

minimum = 0.0
The minimum delay to apply, expressed in seconds.

Return type float

jitter = True
If True, will add a random float between 0 and 1 to the delay.

Return type bool

scale_factor = 1.0
A factor by which the time_to_sleep is multiplied to adjust its scale.

Return type float

time_to_sleep = 0.0 (read-only)
The base number of seconds to delay before allowing a retry.

Return type float

Class Methods

classmethod Polynomial.**delay**(*attempt*, *minimum=None*, *jitter=None*, *scale_factor=1.0*)
Delay for a set period of time based on the attempt.

Parameters

- **attempt** (int) – The number of the attempt that was last-attempted. This value is used by the strategy to determine the amount of time to delay before continuing.
- **minimum** (*number*) – The minimum number of seconds to delay.

  If None, will apply either the strategy's default or the instance's configured property.
- **jitter** (bool) – If True, will add a random float to the delay.

  If False, will not.

  If None, will apply either the strategy's default or the instance's configured property.
- **scale_factor** (float) – A factor by which the time_to_sleep is multiplied to adjust its scale.

  If None, will apply either the strategy's default or the instance's configured property.

4.4 Meta-classes

4.4.1 BackoffStrategy

class BackoffStrategy(*attempt=None*, *minimum=0.0*, *jitter=True*, *scale_factor=1.0*, ***kwargs*)

Abstract Base Class that defines the standard interface exposed by all backoff strategies supported by the library.

Parameters

- **attempt** (int) – The number of the attempt that was last-attempted. This value is used by the strategy to determine the amount of time to delay before continuing.
- **minimum** (*number*) – The minimum delay to apply. Defaults to 0.
- **jitter** (bool) – If True, will add a random float to the delay. Defaults to True.
- **scale_factor** (float) – A factor by which the time_to_sleep is multiplied to adjust its scale. Defaults to 1.0.

Class Attributes

IS_INSTANTIATED = False

Indicates whether the object is an instance of the strategy, or merely its class object.

Return type bool

Properties

attempt = None

The number of the attempt that the strategy is currently evaluating.

Return type int / None

minimum = 0.0

The minimum delay to apply, expressed in seconds.

Return type float

jitter = True

If True, will add a random float between 0 and 1 to the delay.

Return type bool

scale_factor = 1.0
A factor by which the time_to_sleep is multiplied to adjust its scale.

Return type float

time_to_sleep = 0.0 (read-only)
The base number of seconds to delay before allowing a retry.

Return type float

Class Methods

classmethod BackoffStrategy.**delay**(*attempt*, *minimum=None*, *jitter=None*, *scale_factor=1.0*)
Delay for a set period of time based on the attempt.

Parameters

- **attempt** (int) – The number of the attempt that was last-attempted. This value is used by the strategy to determine the amount of time to delay before continuing.
- **minimum** (*number*) – The minimum number of seconds to delay.

  If None, will apply either the strategy's default or the instance's configured property.
- **jitter** (bool) – If True, will add a random float to the delay.

  If False, will not.

  If None, will apply either the strategy's default or the instance's configured property.
- **scale_factor** (float) – A factor by which the time_to_sleep is multiplied to adjust its scale.

  If None, will apply either the strategy's default or the instance's configured property.

CHAPTER 5

Contributing to Backoff-Utils

Note: As a general rule of thumb, the **Backoff-Utils** apply **PEP 8** styling, with some important differences.

Branch	Unit Tests
latest	
v. 1.0	
develop	

What makes an API idiomatic?

One of my favorite ways of thinking about idiomatic design comes from a talk given by Luciano Ramalho at Pycon 2016[5] where he listed traits of a Pythonic API as being:

- don't force [the user] to write boilerplate code
- provide ready to use functions and objects
- don't force [the user] to subclass unless there's a *very good* reason
- include the batteries: make easy tasks easy
- are simple to use but not simplistic: make hard tasks possible
- leverage the Python data model to:
  - provide objects that behave as you expect

- avoid boilerplate through introspection (reflection) and metaprogramming.

Contents:

- *Design Philosophy*
- *Style Guide*
  - *Basic Conventions*
  - *Naming Conventions*
  - *Design Conventions*
  - *Documentation Conventions*
    - *Sphinx*
    - *Docstrings*
- *Dependencies*
- *Preparing Your Development Environment*
- *Ideas and Feature Requests*
- *Testing*
- *Submitting Pull Requests*
- *Building Documentation*
- *References*

5.1 Design Philosophy

The **Backoff-Utils** is meant to be a "beautiful" and "usable" library. That means that it should offer an idiomatic API that:

- works out of the box as intended,
- minimizes "bootstrapping" to produce meaningful output, and
- does not force users to understand how it does what it does.

In other words:

> Users should simply be able to drive the car without looking at the engine.

5.2 Style Guide

5.2.1 Basic Conventions

- Do not terminate lines with semicolons.

[5] https://www.youtube.com/watch?v=k55d3ZUF3ZQ

- Line length should have a maximum of *approximately* 90 characters. If in doubt, make a longer line or break the line between clear concepts.
- Each class should be contained in its own file.
- If a file runs longer than 2,000 lines... it should probably be refactored and split.
- All imports should occur at the top of the file.
- Do not use single-line conditions:

```
# GOOD
if x:
  do_something()

# BAD
if x: do_something()
```

- When testing if an object has a value, be sure to use `if x is None:` or `if x is not None`. Do **not** confuse this with `if x:` and `if not x:`.
- Use the `if x:` construction for testing truthiness, and `if not x:` for testing falsiness. This is **different** from testing:
  - `if x is True:`
  - `if x is False:`
  - `if x is None:`
- As of right now, because we feel that it negatively impacts readability and is less-widely used in the community, we are **not** using type annotations.

5.2.2 Naming Conventions

- `variable_name` and not `variableName` or `VariableName`. Should be a noun that describes what information is contained in the variable. If a `bool`, preface with `is_` or `has_` or similar question-word that can be answered with a yes-or-no.
- `function_name` and not `function_name` or `functionName`. Should be an imperative that describes what the function does (e.g. `get_next_page`).
- `CONSTANT_NAME` and not `constant_name` or `ConstantName`.
- `ClassName` and not `class_name` or `Class_Name`.

5.2.3 Design Conventions

- Functions at the module level can only be aware of objects either at a higher scope or singletons (which effectively have a higher scope).
- Functions and methods can use **one** positional argument (other than `self` or `cls`) without a default value. Any other arguments must be keyword arguments with default value given.

```
def do_some_function(argument):
  # rest of function...

def do_some_function(first_arg,
                     second_arg = None,
```

(continues on next page)

(continued from previous page)

```
                    third_arg = True):
  # rest of function ...
```

- Functions and methods that accept values should start by validating their input, throwing exceptions as appropriate.
- When defining a class, define all attributes in `__init__`.
- When defining a class, start by defining its attributes and methods as private using a single-underscore prefix. Then, only once they're implemented, decide if they should be public.
- Don't be afraid of the private attribute/public property/public setter pattern:

```
class SomeClass(object):
  def __init__(*args, **kwargs):
    self._private_attribute = None

  @property
  def private_attribute(self):
    # custom logic which  may override the default return

    return self._private_attribute

  @setter.private_attribute
  def private_attribute(self, value):
    # custom logic that creates modified_value

    self._private_attribute = modified_value
```

- Separate a function or method's final (or default) `return` from the rest of the code with a blank line (except for single-line functions/methods).

5.2.4 Documentation Conventions

We are very big believers in documentation (maybe you can tell). To document the **Backoff-Utils** we rely on several tools:

Sphinx[1]

Sphinx[1] is used to organize the library's documentation into this lovely readable format (which will also be published to ReadTheDocs[2]). This documentation is written in reStructuredText[3] files which are stored in `<project>/docs`.

Tip: As a general rule of thumb, we try to apply the ReadTheDocs[2] own Documentation Style Guide[4] to our RST documentation.

Hint: To build the HTML documentation locally:

1. In a terminal, navigate to `<project>/docs`.

[1] http://sphinx-doc.org
[2] https://readthedocs.org
[3] http://www.sphinx-doc.org/en/stable/rest.html
[4] http://documentation-style-guide-sphinx.readthedocs.io/en/latest/style-guide.html

2. Execute `make html`.

When built locally, the HTML output of the documentation will be available at `./docs/_build/index.html`.

Docstrings

- Docstrings are used to document the actual source code itself. When writing docstrings we adhere to the conventions outlined in **PEP 257**.

5.3 Dependencies

By design, **Backoff-Utils** are designed to rely on minimal dependencies. The only dependency they have outside of the Python standard library is:

- validator-collection which provides for robust validation functionality.

  This library in turn has two external dependencies: * jsonschema, and * (when installed under Python 2.7) regex

    which is a drop-in replacement for Python 2.7's (buggy) standard `re` module.

5.4 Preparing Your Development Environment

In order to prepare your local development environment, you should:

1. Fork the Git repository.
2. Clone your forked repository.
3. Set up a virtual environment (optional).
4. Install dependencies:

```
backoff-utils/ $ pip install -r requirements.txt
```

And you should be good to go!

5.5 Ideas and Feature Requests

Check for open issues or create a new issue to start a discussion around a bug or feature idea.

5.6 Testing

If you've added a new feature, we recommend you:

- create local unit tests to verify that your feature works as expected, and
- run local unit tests before you submit the pull request to make sure nothing else got broken by accident.

See also:

For more information about the **Backoff-Utils** testing approach please see: *Testing Backoff-Utils*

5.7 Submitting Pull Requests

After you have made changes that you think are ready to be included in the main library, submit a pull request on Github and one of our developers will review your changes. If they're ready (meaning they're well documented, pass unit tests, etc.) then they'll be merged back into the main repository and slated for inclusion in the next release.

5.8 Building Documentation

In order to build documentation locally, you can do so from the command line using:

```
backoff-utils/ $ cd docs
backoff-utils/docs $ make html
```

When the build process has finished, the HTML documentation will be locally available at:

```
backoff-utils/docs/_build/html/index.html
```

Note: Built documentation (the HTML) is **not** included in the project's Git repository. If you need local documentation, you'll need to build it.

5.9 References

CHAPTER 6

Testing the Backoff-Utils

- *Testing Philosophy*
- *Test Organization*
- *Configuring & Running Tests*
  - *Installing with the Test Suite*
  - *Command-line Options*
  - *Configuration File*
  - *Running Tests*
- *Skipping Tests*
- *Incremental Tests*

6.1 Testing Philosophy

Note: Unit tests for the **Backoff-Utils** are written using pytest[1] and a comprehensive set of test automation are provided by tox[2].

There are many schools of thought when it comes to test design. When building the **Backoff-Utils**, we decided to focus on practicality. That means:

- **DRY is good, KISS is better.** To avoid repetition, our test suite makes extensive use of fixtures, parametrization, and decorator-driven behavior. This minimizes the number of test functions that are nearly-identical. However,

[1] https://docs.pytest.org/en/latest/
[2] https://tox.readthedocs.io

there are certain elements of code that are repeated in almost all test functions, as doing so will make future readability and maintenance easier.

- **Coverage matters...kind of.** We have documented the primary intended behavior of every method in the **Backoff-Utils** library, and the most-likely failure modes that can be expected. At the time of writing, we have about 80% code coverage. Yes, yes: We know that is less than 100%. But there are edge cases which are almost impossible to bring about, based on confluences of factors in the wide world. Our goal is to test the key functionality, and as bugs are uncovered to add to the test functions as necessary.

6.2 Test Organization

Each individual test module (e.g. `test_decorator.py`) corresponds to a conceptual grouping of functionality. For example:

- `test_decorator.py` tests the decorator functions found in `backoff_utils/_decorator.py`

Certain test modules may be tightly coupled, as the behavior in one test module may have implications on the execution of tests in another. These test modules use a numbering convention to ensure that they are executed in their required order, so that `test_1_NAME.py` is always executed before `test_2_NAME.py`.

6.3 Configuring & Running Tests

6.3.1 Installing with the Test Suite

Installing via pip

```
$ pip install backoff-utils[tests]
```

From Local Development Environment

See also:

When you *create a local development environment*, all dependencies for running and extending the test suite are installed.

6.3.2 Command-line Options

The **Backoff-Utils** do not use any custom command-line options in their test suite.

Tip: For a full list of the CLI options, including the defaults available, try:

```
backoff-utils $ cd tests/
backoff-utils/tests/ $ pytest --help
```

6.3.3 Configuration File

Because the **Backoff-Utils** has a very simple test suite, we have not prepared a `pytest.ini` configuration file.

6.3.4 Running Tests

Entire Test Suite

```
tests/ $ pytest
```

Test Module

```
tests/ $ pytest tests/test_module.py
```

Test Function

```
tests/ $ pytest tests/test_module.py -k 'test_my_test_function'
```

6.4 Skipping Tests

Note: Because of the simplicity of the **Backoff-Utils**, the test suite does not currently support any test skipping.

6.5 Incremental Tests

Note: The **Backoff-Utils** test suite does support incremental testing using, however at the moment none of the tests designed rely on this functionality.

A variety of test functions are designed to test related functionality. As a result, they are designed to execute incrementally. In order to execute tests incrementally, they need to be defined as methods within a class that you decorate with the `@pytest.mark.incremental` decorator as shown below:

```
@pytest.mark.incremental
class TestIncremental(object):
    def test_function1(self):
        pass
    def test_modification(self):
        assert 0
    def test_modification2(self):
        pass
```

This class will execute the `TestIncremental.test_function1()` test, execute and fail on the `TestIncremental.test_modification()` test, and automatically fail `TestIncremental.test_modification2()` because of the `.test_modification()` failure.

To pass state between incremental tests, add a `state` argument to their method definitions. For example:

```
@pytest.mark.incremental
class TestIncremental(object):
    def test_function(self, state):
        state.is_logged_in = True
        assert state.is_logged_in = True
    def test_modification1(self, state):
        assert state.is_logged_in is True
```

(continues on next page)

(continued from previous page)

```
        state.is_logged_in = False
        assert state.is_logged_in is False
    def test_modification2(self, state):
        assert state.is_logged_in is True
```

Given the example above, the third test (`test_modification2`) will fail because `test_modification` updated the value of `state.is_logged_in`.

Note: `state` is instantiated at the level of the entire test session (one run of the test suite). As a result, it can be affected by tests in other test modules.

CHAPTER 7

Release History

Contents

- *Release History*
  - *Release 1.0.1*
  - *Release 1.0.0*

7.1 Release 1.0.1

- Added changelog to documentation.
- Expanded test matrix to include Python 3.7 and 3.8.
- Removed unnecessary dependencies from `requirements.txt`.

7.2 Release 1.0.0

- First public release of the Backoff-Utils for Python library, with support for five different backoff/retry strategies on Python 2.7, 3.4, 3.5 and 3.6.

CHAPTER 8

Glossary

Backoff Strategy An algorithm that determines how to delay between repeated attempts to perform an operation that has initially failed.

Exponential Backoff A strategy whereby an operation is retried on failure given a randomized delay that raises 2 to the power of the number of attempts that have been made.

Fibonacci Backoff A strategy whereby an operation is retried on failure after delays which grow as per the Fibonacci sequence.

Fixed Backoff A strategy whereby an operation is retried on failure after an explicitly specified delay.

Jitter A random delay between 0 and 1 second in length that can optionally be added to the delay produced by a given backoff strategy.

Linear Backoff A strategy whereby an operation is retried on failure with a monotonic (linearly-growing) delay.

Polynomial Backoff A strategy where an operation is retried on failure with a delay that grows by a factor of the number of attempts that have been made.

Scale Factor The delay determined by the backoff strategy is multipled by this value before the delay is applied. By default, this is set to 1.

Backoff-Utils is a Python library that provides Python functions and decorators that apply various backoff / retry strategies to your Python function and method calls.

The library has a consistent syntax for easy use, and has been tested on Python 2.7, 3.4, 3.5, 3.6, 3.7, and 3.8.

- *Installation*
  - *Importing*
  - *Dependencies*
- *Hello, World Example*
- *Why Backoff-Utils?*

- *Library Features*
  - *Supported Strategies*
  - *Additional Features*
- *Feedback, Support, and Contributing*
- *Indices and tables*

CHAPTER 9

Installation

To install **Backoff-Utils**, just execute:

```
$ pip install backoff-utils
```

9.1 Importing

Once installed, to import **Backoff-Utils** into your project you can use:

```
#: Import the backoff() function.
from backoff_utils import backoff

#: Import the @apply_backoff() decorator.
from backoff_utils import apply_backoff

#: Import backoff strategies.
from backoff_utils import strategies
```

9.2 Dependencies

By design, **Backoff-Utils** are designed to rely on minimal dependencies. The only dependency they have outside of the Python standard library is:

- validator-collection which provides for robust validation functionality.

  This library in turn has two external dependencies: * jsonschema, and * (when installed under Python 2.7) regex

    which is a drop-in replacement for Python 2.7's (buggy) standard `re` module.

CHAPTER 10

Hello, World Example

```
from backoff_utils import strategies

# Using a Function Call
from backoff_utils import backoff

def some_function(arg1, arg2, kwarg1 = None):
    # your code goes here
    pass

result = backoff(some_function,
                 args = ['value1', 'value2'],
                 kwargs = { 'kwarg1': 'value3' },
                 max_tries = 3,
                 max_delay = 3600,
                 strategy = strategies.Exponential)

# Using a Decorator
from backoff_utils import backoff

@apply_backoff(strategy = strategies.Exponential, max_tries = 3, max_delay = 3600)
def some_decorated_function(arg1, arg2, kwarg1 = None):
    # your code goes here
    pass

result = some_decorated_function('value1', 'value2', kwarg1 = 'value3')
```

CHAPTER 11

Why Backoff-Utils?

Because now and again, stuff breaks.

Often, when making external API calls to third-party systems, something goes wrong. The internet might glitch. The API we're calling might timeout. Gremlins might eat our packets. Any number of things can go wrong, and Murphy's law tells us that they will.

Which is why we need *backoff strategies*. Basically, these are techniques that we can use to retry function calls after a given delay - and keep retrying them until either the function call works, or until we've tried so many times that we just give up and handle the error.

This library is meant to be an incredibly simple utility that provides a number of easy-to-use backoff strategies. Its core API is to expose:

- the `backoff()` function, which lets you apply a given backoff strategy to any Python function call, and;
- the `@apply_backoff()` decorator, which lets you decorate any function or method call so that a given backoff strategy is *always* applied when the decorated function/method is called.

See also:

For more information about how to use the library, please see *Using the Library*

CHAPTER 12

Library Features

12.1 Supported Strategies

The library supports five of the most-common backoff strategies that we've come across:

- Exponential
- Fibonacci
- Fixed
- Linear
- Polynomial

In addtion, you can also *create your own custom strategies* as well.

See also:

For more information about the backoff strategies supported, please see: *Strategies Explained*

12.2 Additional Features

In addition to the basic strategies, the library also supports:

- *random jitter*
- *argument-adjustment on retry*
- *selective exception capture*
- *chained backoff strategies*
- *failure handlers*
- *success handlers*
- *cut-off after a max delay*

- *cut-off after max tries*
- *scaling*
- *minimum delay*

CHAPTER 13

Feedback, Support, and Contributing

We're happy to maintain this library going forward, and would always love to hear users' feedback - especially if you're running into issues.

Please report issues or questions on the project's Github page

We also welcome community contributions - for more information, please see the *Contributor Guide* .

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

backoff_utils.strategies, 27

t

tests, 43

Index

A

apply_backoff() (*in module backoff_utils._decorator*), 25
attempt (*in module backoff_utils.strategies*), 27, 28, 30, 31, 33, 34

B

Backoff Strategy, **49**
backoff() (*in module backoff_utils._backoff*), 23
backoff_utils.strategies (*module*), 27
BackoffStrategy (*class in backoff_utils.strategies*), 34

D

delay() (*backoff_utils.strategies.BackoffStrategy class method*), 35
delay() (*backoff_utils.strategies.Exponential class method*), 28
delay() (*backoff_utils.strategies.Fibonacci class method*), 29
delay() (*backoff_utils.strategies.Fixed class method*), 31
delay() (*backoff_utils.strategies.Linear class method*), 32
delay() (*backoff_utils.strategies.Polynomial class method*), 33

E

exponent (*in module backoff_utils.strategies*), 33
Exponential (*class in backoff_utils.strategies*), 27
Exponential Backoff, **49**

F

Fibonacci (*class in backoff_utils.strategies*), 28
Fibonacci Backoff, **49**
Fixed (*class in backoff_utils.strategies*), 29
Fixed Backoff, **49**

I

IS_INSTANTIATED (*in module backoff_utils.strategies*), 27, 28, 30, 31, 33, 34

J

Jitter, **49**
jitter (*in module backoff_utils.strategies*), 27, 29, 30, 32–34

L

Linear (*class in backoff_utils.strategies*), 31
Linear Backoff, **49**

M

minimum (*in module backoff_utils.strategies*), 27, 28, 30, 31, 33, 34

P

Polynomial (*class in backoff_utils.strategies*), 32
Polynomial Backoff, **49**
Python Enhancement Proposals
    PEP 257, 41
    PEP 8, 37

S

Scale Factor, **49**
scale_factor (*in module backoff_utils.strategies*), 27, 29, 30, 32, 33, 35
sequence (*in module backoff_utils.strategies*), 30

T

tests (*module*), 43
time_to_sleep (*in module backoff_utils.strategies*), 27, 29, 30, 32, 33, 35