Python Backoff Module

Writing a function which can have intermittent failures? Or calling an external API and want to retry the call in case of failures?

Retry when exception happens

The `on_exception` decorator is used to retry when a specified exception is raised. Here's an example using exponential backoff when any `requests` exception is raised:

```
@backoff.on_exception(backoff.expo,
                      requests.exceptions.RequestException)
def get_url(url):
    return requests.get(url)
```

The decorator will also accept a tuple of exceptions for cases where the same backoff behavior is desired for more than one exception type:

```
@backoff.on_exception(backoff.expo,
                      (requests.exceptions.Timeout,
                       requests.exceptions.ConnectionError))
def get_url(url):
    return requests.get(url)
```

Define when to give up retrying

The keyword argument `max_time` specifies the maximum amount of total time in seconds that can elapse before giving up.

```
@backoff.on_exception(backoff.expo,
                      requests.exceptions.RequestException,
                      max_time=60)
def get_url(url):
    return requests.get(url)
```

Keyword argument `max_tries` specifies the maximum number of calls to make to the target function before giving up.

```
@backoff.on_exception(backoff.expo,
                      requests.exceptions.RequestException,
                      max_tries=8,
                      jitter=None)
def get_url(url):
    return requests.get(url)
```

Define when to give up retrying

In some cases the raised exception instance itself may need to be inspected in order to determine if it is a retryable condition. The `giveup` keyword arg can be used to specify a function which accepts the exception and returns a truthy value if the exception should not be retried:

```
def fatal_code(e):
    return 400 <= e.response.status_code < 500

@backoff.on_exception(backoff.expo,
                      requests.exceptions.RequestException,
                      max_time=300,
                      giveup=fatal_code)
def get_url(url):
    return requests.get(url)
```

Use it for polling external resources till you get a result

@backoff.on_predicate

The `on_predicate` decorator is used to retry when a particular condition is true of the return value of the target function. This may be useful when polling a resource for externally generated content.

Here's an example which uses a fibonacci sequence backoff when the return value of the target function is the empty list:

```
@backoff.on_predicate(backoff.fibo, lambda x: x == [], max_value=13)
def poll_for_messages(queue):
    return queue.get()
```

Use it for polling external resources till you get a result

More simply, a function which continues polling every second until it gets a non-falsey result could be defined like like this:

```
@backoff.on_predicate(backoff.constant, jitter=None, interval=1)
def poll_for_message(queue):
    return queue.get()
```

Use multiple decorators to handle complex scenarios

In this example we are polling a queue till we get falsey result (empty value). We will stop polling when it returns a value. While calling queue.get( ) we are implementing separate retry logic in case of HTTP exceptions and Timeout errors.

The backoff decorators may also be combined to specify different backoff behavior for different cases:

```
@backoff.on_predicate(backoff.fibo, max_value=13)
@backoff.on_exception(backoff.expo,
                      requests.exceptions.HTTPError,
                      max_time=60)
@backoff.on_exception(backoff.expo,
                      requests.exceptions.Timeout,
                      max_time=300)
def poll_for_message(queue):
    return queue.get()
```