vcrpy Documentation

Release 1.7.4

Kevin McCarthy

Nov 03, 2019

Contents

1 Rationale **3**

2 License **5**

3 Contents **7**

3.1 Installation . . . 7
3.2 Usage . . . 8
3.3 Configuration . . . 10
3.4 Advanced Features . . . 11
3.5 API . . . 17
3.6 Debugging . . . 19
3.7 CannotOverwriteExistingCassetteException . . . 20
3.8 Contributing . . . 21
3.9 Changelog . . . 22

Python Module Index **27**

Index **29**

This is a Python version of Ruby's VCR library.

Source code https://github.com/kevin1024/vcrpy

Documentation https://vcrpy.readthedocs.io/

CHAPTER 1

Rationale

VCR.py simplifies and speeds up tests that make HTTP requests. The first time you run code that is inside a VCR.py context manager or decorated function, VCR.py records all HTTP interactions that take place through the libraries it supports and serializes and writes them to a flat file (in yaml format by default). This flat file is called a cassette. When the relevant piece of code is executed again, VCR.py will read the serialized requests and responses from the aforementioned cassette file, and intercept any HTTP requests that it recognizes from the original test run and return the responses that corresponded to those requests. This means that the requests will not actually result in HTTP traffic, which confers several benefits including:

- The ability to work offline
- Completely deterministic tests
- Increased test execution speed

If the server you are testing against ever changes its API, all you need to do is delete your existing cassette files, and run your tests again. VCR.py will detect the absence of a cassette file and once again record all HTTP interactions, which will update them to correspond to the new API.

CHAPTER 2

License

This library uses the MIT license. See LICENSE.txt for more details

CHAPTER 3

Contents

3.1 Installation

VCR.py is a package on PyPI, so you can install with pip:

```
pip install vcrpy
```

3.1.1 Compatibility

VCR.py supports Python 2.7 and 3.5+, and pypy.

The following HTTP libraries are supported:

- `aiohttp`
- `boto`
- `boto3`
- `http.client`
- `httplib2`
- `requests` (both 1.x and 2.x versions)
- `tornado.httpclient`
- `urllib2`
- `urllib3`

3.1.2 Speed

VCR.py runs about 10x faster when pyyaml can use the libyaml extensions. In order for this to work, libyaml needs to be available when pyyaml is built. Additionally the flag is cached by pip, so you might need to explicitly avoid the

cache when rebuilding pyyaml.

1. Test if pyyaml is built with libyaml. This should work:

```
python -c 'from yaml import CLoader'
```

2. Install libyaml according to your Linux distribution, or using Homebrew on Mac:

```
brew install libyaml          # Mac with Homebrew
apt-get install libyaml-dev   # Ubuntu
dnf install libyaml-devel       # Fedora
```

3. Rebuild pyyaml with libyaml:

```
pip uninstall pyyaml
pip --no-cache-dir install pyyaml
```

3.1.3 Upgrade

New Cassette Format

The cassette format has changed in *VCR.py 1.x*, the *VCR.py 0.x* cassettes cannot be used with *VCR.py 1.x*. The easiest way to upgrade is to simply delete your cassettes and re-record all of them. VCR.py also provides a migration script that attempts to upgrade your 0.x cassettes to the new 1.x format. To use it, run the following command:

```
python -m vcr.migration PATH
```

The PATH can be either a path to the directory with cassettes or the path to a single cassette.

Note: Back up your cassettes files before migration. The migration *should* only modify cassettes using the old 0.x format.

New serializer / deserializer API

If you made a custom serializer, you will need to update it to match the new API in version 1.0.x

- Serializers now take dicts and return strings.
- Deserializers take strings and return dicts (instead of requests, responses pair)

3.1.4 Ruby VCR compatibility

VCR.py does not aim to match the format of the Ruby VCR YAML files. Cassettes generated by Ruby's VCR are not compatible with VCR.py.

3.2 Usage

```
import vcr
import urllib2

with vcr.use_cassette('fixtures/vcr_cassettes/synopsis.yaml'):
    response = urllib2.urlopen('http://www.iana.org/domains/reserved').read()
    assert 'Example domains' in response
```

Run this test once, and VCR.py will record the HTTP request to `fixtures/vcr_cassettes/synopsis.yaml`. Run it again, and VCR.py will replay the response from iana.org when the http request is made. This test is now fast (no real HTTP requests are made anymore), deterministic (the test will continue to pass, even if you are offline, or iana.org goes down for maintenance) and accurate (the response will contain the same headers and body you get from a real request).

You can also use VCR.py as a decorator. The same request above would look like this:

```
@vcr.use_cassette('fixtures/vcr_cassettes/synopsis.yaml')
def test_iana():
    response = urllib2.urlopen('http://www.iana.org/domains/reserved').read()
    assert 'Example domains' in response
```

When using the decorator version of `use_cassette`, it is possible to omit the path to the cassette file.

```
@vcr.use_cassette()
def test_iana():
    response = urllib2.urlopen('http://www.iana.org/domains/reserved').read()
    assert 'Example domains' in response
```

In this case, the cassette file will be given the same name as the test function, and it will be placed in the same directory as the file in which the test is defined. See the Automatic Test Naming section below for more details.

3.2.1 Record Modes

VCR supports 4 record modes (with the same behavior as Ruby's VCR):

once

- Replay previously recorded interactions.
- Record new interactions if there is no cassette file.
- Cause an error to be raised for new requests if there is a cassette file.

It is similar to the new_episodes record mode, but will prevent new, unexpected requests from being made (e.g. because the request URI changed).

once is the default record mode, used when you do not set one.

new_episodes

- Record new interactions.
- Replay previously recorded interactions. It is similar to the once record mode, but will always record new interactions, even if you have an existing recorded one that is similar, but not identical.

This was the default behavior in versions < 0.3.0

none

- Replay previously recorded interactions.
- Cause an error to be raised for any new requests. This is useful when your code makes potentially dangerous HTTP requests. The none record mode guarantees that no new HTTP requests will be made.

all

- Record new interactions.
- Never replay previously recorded interactions. This can be temporarily used to force VCR to re-record a cassette (i.e. to ensure the responses are not out of date) or can be used when you simply want to log all HTTP requests.

3.2.2 Unittest Integration

While it's possible to use the context manager or decorator forms with unittest, there's also a `VCRTestCase` provided separately by vcrpy-unittest.

3.2.3 Pytest Integration

A Pytest plugin is available here : pytest-vcr.

Alternative plugin, that also provides network access blocking: pytest-recording.

3.3 Configuration

If you don't like VCR's defaults, you can set options by instantiating a `VCR` class and setting the options on it.

```
import vcr

my_vcr = vcr.VCR(
    serializer='json',
    cassette_library_dir='fixtures/cassettes',
    record_mode='once',
    match_on=['uri', 'method'],
)

with my_vcr.use_cassette('test.json'):
    # your http code here
```

Otherwise, you can override options each time you use a cassette.

```
with vcr.use_cassette('test.yml', serializer='json', record_mode='once'):
    # your http code here
```

Note: Per-cassette overrides take precedence over the global config.

3.3.1 Request matching

Request matching is configurable and allows you to change which requests VCR considers identical. The default behavior is `['method', 'scheme', 'host', 'port', 'path', 'query']` which means that requests with both the same URL and method (ie POST or GET) are considered identical.

This can be configured by changing the `match_on` setting.

The following options are available :

- method (for example, POST or GET)
- uri (the full URI.)

- host (the hostname of the server receiving the request)
- port (the port of the server receiving the request)
- path (the path of the request)
- query (the query string of the request)
- raw_body (the entire request body as is)
- body (the entire request body unmarshalled by content-type i.e. xmlrpc, json, form-urlencoded, falling back on raw_body)
- headers (the headers of the request)

  Backwards compatible matchers:

- url (the `uri` alias)

If these options don't work for you, you can also register your own request matcher. This is described in the Advanced section of this README.

3.4 Advanced Features

If you want, VCR.py can return information about the cassette it is using to record your requests and responses. This will let you record your requests and responses and make assertions on them, to make sure that your code under test is generating the expected requests and responses. This feature is not present in Ruby's VCR, but I think it is a nice addition. Here's an example:

```
import vcr
import urllib2

with vcr.use_cassette('fixtures/vcr_cassettes/synopsis.yaml') as cass:
    response = urllib2.urlopen('http://www.zombo.com/').read()
    # cass should have 1 request inside it
    assert len(cass) == 1
    # the request uri should have been http://www.zombo.com/
    assert cass.requests[0].uri == 'http://www.zombo.com/'
```

The `Cassette` object exposes the following properties which I consider part of the API. The fields are as follows:

- `requests`: A list of vcr.Request objects corresponding to the http requests that were made during the recording of the cassette. The requests appear in the order that they were originally processed.
- `responses`: A list of the responses made.
- `play_count`: The number of times this cassette has played back a response.
- `all_played`: A boolean indicating whether all the responses have been played back.
- `responses_of(request)`: Access the responses that match a given request

The `Request` object has the following properties:

- `uri`: The full uri of the request. Example: "https://google.com/?q=vcrpy"
- `scheme`: The scheme used to make the request (http or https)
- `host`: The host of the request, for example "www.google.com"
- `port`: The port the request was made on
- `path`: The path of the request. For example "/" or "/home.html"

- `query`: The parsed query string of the request. Sorted list of name, value pairs.
- `method` : The method used to make the request, for example "GET" or "POST"
- `body`: The body of the request, usually empty except for POST / PUT / etc

Backwards compatible properties:

- `url`: The `uri` alias
- `protocol`: The `scheme` alias

3.4.1 Register your own serializer

Don't like JSON or YAML? That's OK, VCR.py can serialize to any format you would like. Create your own module or class instance with 2 methods:

- `def deserialize(cassette_string)`
- `def serialize(cassette_dict)`

Finally, register your class with VCR to use your new serializer.

```
import vcr

class BogoSerializer(object):
    """
    Must implement serialize() and deserialize() methods
    """
    pass

my_vcr = vcr.VCR()
my_vcr.register_serializer('bogo', BogoSerializer())

with my_vcr.use_cassette('test.bogo', serializer='bogo'):
    # your http here

# After you register, you can set the default serializer to your new serializer

my_vcr.serializer = 'bogo'

with my_vcr.use_cassette('test.bogo'):
    # your http here
```

3.4.2 Register your own request matcher

Create your own method with the following signature

```
def my_matcher(r1, r2):
```

Your method receives the two requests and can either:

- Use an `assert` statement: return None if they match and raise `AssertionError` if not.
- Return a boolean: `True` if they match, `False` if not.

Note: in order to have good feedback when a matcher fails, we recommend using an `assert` statement with a clear error message.

Finally, register your method with VCR to use your new request matcher.

```
import vcr

def jurassic_matcher(r1, r2):
    assert r1.uri == r2.uri and 'JURASSIC PARK' in r1.body, \
        'required string (JURASSIC PARK) not found in request body'

my_vcr = vcr.VCR()
my_vcr.register_matcher('jurassic', jurassic_matcher)

with my_vcr.use_cassette('test.yml', match_on=['jurassic']):
    # your http here

# After you register, you can set the default match_on to use your new matcher

my_vcr.match_on = ['jurassic']

with my_vcr.use_cassette('test.yml'):
    # your http here
```

3.4.3 Register your own cassette persister

Create your own persistence class, see the persister_example.

Your custom persister must implement both `load_cassette` and `save_cassette` methods. The `load_cassette` method must return a deserialized cassette or raise

`ValueError` if no cassette is found.

Once the persister class is defined, register with VCR like so…

```
import vcr
my_vcr = vcr.VCR()

class CustomerPersister(object):
    # implement Persister methods...

my_vcr.register_persister(CustomPersister)
```

3.4.4 Filter sensitive data from the request

If you are checking your cassettes into source control, and are using some form of authentication in your tests, you can filter out that information so it won't appear in your cassette files. There are a few ways to do this:

Filter information from HTTP Headers

Use the `filter_headers` configuration option with a list of headers to filter.

```
with my_vcr.use_cassette('test.yml', filter_headers=['authorization']):
    # sensitive HTTP request goes here
```

Filter information from HTTP querystring

Use the `filter_query_parameters` configuration option with a list of query parameters to filter.

```
with my_vcr.use_cassette('test.yml', filter_query_parameters=['api_key']):
    requests.get('http://api.com/getdata?api_key=secretstring')
```

Filter information from HTTP post data

Use the `filter_post_data_parameters` configuration option with a list of post data parameters to filter.

```
with my_vcr.use_cassette('test.yml', filter_post_data_parameters=['client_secret']):
    requests.post('http://api.com/postdata', data={'api_key': 'secretstring'})
```

Advanced use of filter_headers, filter_query_parameters and filter_post_data_parameters

In all of the above cases, it's also possible to pass a list of `(key, value)` tuples where the value can be any of the following:

- A new value to replace the original value.
- `None` to remove the key/value pair. (Same as passing a simple key string.)
- A callable that returns a new value or `None`.

So these two calls are the same:

```
# original (still works)
vcr = VCR(filter_headers=['authorization'])

# new
vcr = VCR(filter_headers=[('authorization', None)])
```

Here are two examples of the new functionality:

```
# replace with a static value (most common)
vcr = VCR(filter_headers=[('authorization', 'XXXXXX')])

# replace with a callable, for example when testing
# lots of different kinds of authorization.
def replace_auth(key, value, request):
    auth_type = value.split(' ', 1)[0]
    return '{} {}'.format(auth_type, 'XXXXXX')
```

Custom Request filtering

If none of these covers your request filtering needs, you can register a callback with the `before_record_request` configuration option to manipulate the HTTP request before adding it to the cassette, or return `None` to ignore it entirely. Here is an example that will never record requests to the `'/login'` path:

```
def before_record_cb(request):
    if request.path == '/login':
        return None
    return request

my_vcr = vcr.VCR(
    before_record_request=before_record_cb,
)
```

(continues on next page)

(continued from previous page)

```
with my_vcr.use_cassette('test.yml'):
    # your http code here
```

You can also mutate the request using this callback. For example, you could remove all query parameters from any requests to the `'/login'` path.

```
def scrub_login_request(request):
    if request.path == '/login':
        request.uri, _ =  urllib.splitquery(request.uri)
    return request

my_vcr = vcr.VCR(
    before_record_request=scrub_login_request,
)
with my_vcr.use_cassette('test.yml'):
    # your http code here
```

Custom Response Filtering

You can also do response filtering with the `before_record_response` configuration option. Its usage is similar to the above `before_record_request` - you can mutate the response, or return `None` to avoid recording the request and response altogether. For example to hide sensitive data from the request body:

```
def scrub_string(string, replacement=''):
    def before_record_response(response):
        response['body']['string'] = response['body']['string'].replace(string,␣
↪replacement)
        return response
    return before_record_response

my_vcr = vcr.VCR(
    before_record_response=scrub_string(settings.USERNAME, 'username'),
)
with my_vcr.use_cassette('test.yml'):
     # your http code here
```

3.4.5 Decode compressed response

When the `decode_compressed_response` keyword argument of a `VCR` object is set to True, VCR will decompress "gzip" and "deflate" response bodies before recording. This ensures that these interactions become readable and editable after being serialized.

Note: Decompression is done before any other specified *Custom Response Filtering*.

This option should be avoided if the actual decompression of response bodies is part of the functionality of the library or app being tested.

3.4.6 Ignore requests

If you would like to completely ignore certain requests, you can do it in a few ways:

- Set the `ignore_localhost` option equal to True. This will not record any requests sent to (or responses from) localhost, 127.0.0.1, or 0.0.0.0.
- Set the `ignore_hosts` configuration option to a list of hosts to ignore
- Add a `before_record_request` or `before_record_response` callback that returns `None` for requests you want to ignore (see above).

Requests that are ignored by VCR will not be saved in a cassette, nor played back from a cassette. VCR will completely ignore those requests as if it didn't notice them at all, and they will continue to hit the server as if VCR were not there.

3.4.7 Custom Patches

If you use a custom `HTTPConnection` class, or otherwise make http requests in a way that requires additional patching, you can use the `custom_patches` keyword argument of the `VCR` and `Cassette` objects to patch those objects whenever a cassette's context is entered. To patch a custom version of `HTTPConnection` you can do something like this:

```
import where_the_custom_https_connection_lives
from vcr.stubs import VCRHTTPSConnection
my_vcr = config.VCR(custom_patches=((where_the_custom_https_connection_lives,
↪'CustomHTTPSConnection', VCRHTTPSConnection),))

@my_vcr.use_cassette(...)
```

3.4.8 Automatic Cassette Naming

VCR.py now allows the omission of the path argument to the use_cassette function. Both of the following are now legal/should work

```
@my_vcr.use_cassette
def my_test_function():
    ...
```

```
@my_vcr.use_cassette()
def my_test_function():
    ...
```

In both cases, VCR.py will use a path that is generated from the provided test function's name. If no `cassette_library_dir` has been set, the cassette will be in a file with the name of the test function in directory of the file in which the test function is declared. If a `cassette_library_dir` has been set, the cassette will appear in that directory in a file with the name of the decorated function.

It is possible to control the path produced by the automatic naming machinery by customizing the `path_transformer` and `func_path_generator` vcr variables. To add an extension to all cassette names, use `VCR.ensure_suffix` as follows:

```
my_vcr = VCR(path_transformer=VCR.ensure_suffix('.yaml'))

@my_vcr.use_cassette
def my_test_function():
```

3.4.9 Rewind Cassette

VCR.py allows to rewind a cassette in order to replay it inside the same function/test.

```
with vcr.use_cassette('fixtures/vcr_cassettes/synopsis.yaml') as cass:
    response = urllib2.urlopen('http://www.zombo.com/').read()
    assert cass.all_played
    a.rewind()
    assert not cass.all_played
```

3.5 API

3.5.1 config

3.5.2 cassette

class vcr.cassette.**Cassette**(*path*, *serializer=None*, *persister=None*, *record_mode='once'*, *match_on=(<function uri>, <function method>)*, *before_record_request=None*, *before_record_response=None*, *custom_patches=()*, *inject=False*)

A container for recorded requests and responses

__init__(*path*, *serializer=None*, *persister=None*, *record_mode='once'*, *match_on=(<function uri>, <function method>)*, *before_record_request=None*, *before_record_response=None*, *custom_patches=()*, *inject=False*)

Initialize self. See help(type(self)) for accurate signature.

all_played

Returns True if all responses have been played, False otherwise.

append(*request*, *response*)

Add a request, response pair to this cassette

find_requests_with_most_matches(*request*)

Get the most similar request(s) stored in the cassette of a given request as a list of tuples like this: - the request object - the successful matchers as string - the failed matchers and the related assertion message with the difference details as strings tuple

This is useful when a request failed to be found, we can get the similar request(s) in order to know what have changed in the request parts.

classmethod load(***kwargs*)

Instantiate and load the cassette stored at the specified path.

play_response(*request*)

Get the response corresponding to a request, but only if it hasn't been played back before, and mark it as played

responses_of(*request*)

Find the responses corresponding to a request. This function isn't actually used by VCR internally, but is provided as an external API.

class vcr.cassette.**CassetteContextDecorator**(*cls*, *args_getter*)

Context manager/decorator that handles installing the cassette and removing cassettes.

This class defers the creation of a new cassette instance until the point at which it is installed by context manager or decorator. The fact that a new cassette is used with each application prevents the state of any cassette from interfering with another.

Instances of this class are NOT reentrant as context managers. However, functions that are decorated by `CassetteContextDecorator` instances ARE reentrant. See the implementation of `__call__` on this class for more details. There is also a guard against attempts to reenter instances of this class as a context manager in `__exit__`.

__init__(*cls*, *args_getter*)
Initialize self. See help(type(self)) for accurate signature.

3.5.3 matchers

vcr.matchers.**get_assertion_message**(*assertion_details*)
Get a detailed message about the failing matcher.

vcr.matchers.**get_matchers_results**(*r1*, *r2*, *matchers*)
Get the comparison results of two requests as two list. The first returned list represents the matchers names that passed. The second list is the failed matchers as a string with failed assertion details if any.

3.5.4 filters

vcr.filters.**decode_response**(*response*)

If the response is compressed with gzip or deflate:

1. decompress the response body
2. delete the content-encoding header
3. update content-length header to decompressed length

vcr.filters.**remove_headers**(*request*, *headers_to_remove*)
Wrap replace_headers() for API backward compatibility.

vcr.filters.**remove_post_data_parameters**(*request*, *post_data_parameters_to_remove*)
Wrap replace_post_data_parameters() for API backward compatibility.

vcr.filters.**remove_query_parameters**(*request*, *query_parameters_to_remove*)
Wrap replace_query_parameters() for API backward compatibility.

vcr.filters.**replace_headers**(*request*, *replacements*)
Replace headers in request according to replacements. The replacements should be a list of (key, value) pairs where the value can be any of:

1. A simple replacement string value.
2. None to remove the given header.
3. A callable which accepts (key, value, request) and returns a string value or None.

vcr.filters.**replace_post_data_parameters**(*request*, *replacements*)
Replace post data in request–either form data or json–according to replacements. The replacements should be a list of (key, value) pairs where the value can be any of:

1. A simple replacement string value.
2. None to remove the given header.
3. A callable which accepts (key, value, request) and returns a string value or None.

`vcr.filters.`**`replace_query_parameters`**(*request*, *replacements*)

Replace query parameters in request according to replacements. The replacements should be a list of (key, value) pairs where the value can be any of:

1. A simple replacement string value.
2. None to remove the given header.
3. A callable which accepts (key, value, request) and returns a string value or None.

3.5.5 request

class `vcr.request.`**`HeadersDict`**(*data=None*, ***kwargs*)

There is a weird quirk in HTTP. You can send the same header twice. For this reason, headers are represented by a dict, with lists as the values. However, it appears that HTTPlib is completely incapable of sending the same header twice. This puts me in a weird position: I want to be able to accurately represent HTTP headers in cassettes, but I don't want the extra step of always having to do [0] in the general case, i.e. request.headers['key'][0]

In addition, some servers sometimes send the same header more than once, and httplib *can* deal with this situation.

Furthermore, I wanted to keep the request and response cassette format as similar as possible.

For this reason, in cassettes I keep a dict with lists as keys, but once deserialized into VCR, I keep them as plain, naked dicts.

class `vcr.request.`**`Request`**(*method*, *uri*, *body*, *headers*)

VCR's representation of a request.

`__init__`(*method*, *uri*, *body*, *headers*)

Initialize self. See help(type(self)) for accurate signature.

3.5.6 serialize

`vcr.serialize.`**`CASSETTE_FORMAT_VERSION = 1`**

Just a general note on the serialization philosophy here: I prefer cassettes to be human-readable if possible. Yaml serializes bytestrings to !!binary, which isn't readable, so I would like to serialize to strings and from strings, which yaml will encode as utf-8 automatically. All the internal HTTP stuff expects bytestrings, so this whole serialization process feels backwards.

Serializing: bytestring -> string (yaml persists to utf-8) Deserializing: string (yaml converts from utf-8) -> bytestring

3.5.7 patch

Utilities for patching in cassettes

3.6 Debugging

VCR.py has a few log messages you can turn on to help you figure out if HTTP requests are hitting a real server or not. You can turn them on like this:

```
import vcr
import requests
import logging

logging.basicConfig() # you need to initialize logging, otherwise you will not see␣
↪anything from vcrpy
vcr_log = logging.getLogger("vcr")
vcr_log.setLevel(logging.INFO)

with vcr.use_cassette('headers.yml'):
    requests.get('http://httpbin.org/headers')
```

The first time you run this, you will see:

```
INFO:vcr.stubs:<Request (GET) http://httpbin.org/headers> not in cassette, sending to␣
↪real server
```

The second time, you will see:

```
INFO:vcr.stubs:Playing response for <Request (GET) http://httpbin.org/headers> from␣
↪cassette
```

If you set the loglevel to DEBUG, you will also get information about which matchers didn't match. This can help you with debugging custom matchers.

3.7 CannotOverwriteExistingCassetteException

When a request failed to be found in an existing cassette, VCR.py tries to get the request(s) that may be similar to the one being searched. The goal is to see which matcher(s) failed and understand what part of the failed request may have changed. It can return multiple similar requests with :

- the matchers that have succeeded
- the matchers that have failed
- for each failed matchers, why it has failed with an assertion message

CannotOverwriteExistingCassetteException message example :

```
CannotOverwriteExistingCassetteException: Can't overwrite existing cassette (
↪'cassette.yaml') in your current record mode ('once').
No match for the request (<Request (GET) https://www.googleapis.com/?alt=json&
↪maxResults=200>) was found.
Found 1 similar requests with 1 different matchers :

1 - (<Request (GET) https://www.googleapis.com/?alt=json&maxResults=500>).
Matchers succeeded : ['method', 'scheme', 'host', 'port', 'path']
Matchers failed :
query - assertion failure :
[('alt', 'json'), ('maxResults', '200')] != [('alt', 'json'), ('maxResults', '500')]
```

3.8 Contributing

3.8.1 Running VCR's test suite

The tests are all run automatically on Travis CI, but you can also run them yourself using py.test and Tox. Tox will automatically run them in all environments VCR.py supports. The test suite is pretty big and slow, but you can tell tox to only run specific tests like this:

```
tox -e {pyNN}-{HTTP_LIBRARY} -- <pytest flags passed through>

tox -e py27-requests -- -v -k "'test_status_code or test_gzip'"
tox -e py37-requests -- -v --last-failed
```

This will run only tests that look like `test_status_code` or `test_gzip` in the test suite, and only in the python 2.7 environment that has `requests` installed.

Also, in order for the boto tests to run, you will need an AWS key. Refer to the boto documentation for how to set this up. I have marked the boto tests as optional in Travis so you don't have to worry about them failing if you submit a pull request.

3.8.2 Using PyEnv with VCR's test suite

PyEnv is a tool for managing multiple installation of python on your system. See the full documentation at their *github <https://github.com/pyenv/pyenv>* but we are also going to use *tox-pyenv <https://pypi.org/project/tox-pyenv/>* in this example:

```
git clone https://github.com/pyenv/pyenv ~/.pyenv

# Add ~/.pyenv/bin to your PATH
export PATH="$PATH:~/.pyenv/bin"

# Setup shim paths
eval "$(pyenv init -)"

# Setup your local system tox tooling
pip install tox tox-pyenv

# Install supported versions (at time of writing), this does not activate them
pyenv install 2.7.10 3.5.7 3.6.9 3.7.4 3.8-dev pypy2.6-7.1.1 pypy3.6-7.1.1

# This activates them
pyenv local 2.7.10 3.5.7 3.6.9 3.7.4 3.8-dev pypy2.6-7.1.1 pypy3.6-7.1.1

# Run the whole test suite
tox

# Run the whole test suite or just part of it
tox -e lint
tox -e py37-requests
```

3.8.3 Troubleshooting on MacOSX

If you have this kind of error when running tox :

```
__main__.ConfigurationError: Curl is configured to use SSL, but we have
not been able to determine which SSL backend it is using. Please see PycURL␣
↪documentation for how to specify the SSL backend manually.
```

Then you need to define some environment variables:

```
export PYCURL_SSL_LIBRARY=openssl
export LDFLAGS=-L/usr/local/opt/openssl/lib
export CPPFLAGS=-I/usr/local/opt/openssl/include
```

Reference : stackoverflow issue

3.9 Changelog

- 2.1.x (UNRELEASED)
-
- 2.1.1
- Format code with black (@neozenith)
- Use latest pypy3 in Travis (@hugovk)
- Improve documentation about custom matchers (@gward)
- Fix exception when body is empty (@keithprickett)
- Add *pytest-recording* to the documentation as an alternative Pytest plugin (@Stranger6667)
- Fix yarl and python3.5 version issue (@neozenith)
- Fix header matcher for boto3 - fixes #474 (@simahawk)
- 2.1.0 - Add a *rewind* method to reset a cassette (thanks @khamidou) New error message with more details on why the cassette failed to play a request (thanks @arthurHamon2, @neozenith) Handle connect tunnel URI (thanks @jeking3) Add code coverage to the project (thanks @neozenith) Drop support to python 3.4 Add deprecation warning on python 2.7, next major release will drop python 2.7 support Fix build problems on requests tests (thanks to @dunossauro) Fix matching on 'body' failing when Unicode symbols are present in them (thanks @valgur) Fix bugs on aiohttp integration (thanks @graingert, @steinnes, @stj, @lamenezes, @lmazuel) Fix Biopython incompatibility (thanks @rishab121) Fix Boto3 integration (thanks @1oglop1, @arthurHamon2)
- 2.0.1 - Fix bug when using vcrpy with python 3.4
- 2.0.0 - Support python 3.7 (fix httplib2 and urllib2, thanks @felixonmars) [#356] Fixes *before_record_response* so the original response isn't changed (thanks @kgraves) Fix requests stub when using proxy (thanks @samuelfekete @daneoshiga) (only for aiohttp stub) Drop support to python 3.4 asyncio.coroutine (aiohttp doesn't support python it anymore) Fix aiohttp stub to work with aiohttp client (thanks @stj) Fix aiohttp stub to accept content type passed Improve docs (thanks @adamchainz)
- 1.13.0 - Fix support to latest aiohttp version (3.3.2). Fix content-type bug in aiohttp stub. Save URL with query params properly when using aiohttp.
- 1.12.0 - Fix support to latest aiohttp version (3.2.1), Adapted setup to PEP508, Support binary responses on aiohttp, Dropped support for EOL python versions (2.6 and 3.3)
- 1.11.1 Fix compatibility with newest requests and urllib3 releases
- 1.11.0 Allow injection of persistence methods + bugfixes (thanks @j-funk and @IvanMalison), Support python 3.6 + CI tests (thanks @derekbekoe and @graingert), Support pytest-asyncio coroutines (thanks @graingert)

- 1.10.5 Added a fix to httplib2 (thanks @carlosds730), Fix an issue with aiohttp (thanks @madninja), Add missing requirement yarl (thanks @lamenezes), Remove duplicate mock triple (thanks @FooBarQuaxx)
- 1.10.4 Fix an issue with asyncio aiohttp (thanks @madninja)
- 1.10.3 Fix some issues with asyncio and params (thanks @anovikov1984 and @lamenezes), Fix some issues with cassette serialize / deserialize and empty response bodies (thanks @gRoussac and @dz0ny)
- 1.10.2 Fix 1.10.1 release - add aiohttp support back in
- 1.10.1 [bad release] Fix build for Fedora package + python2 (thanks @puiterwijk and @lamenezes)
- 1.10.0 Add support for aiohttp (thanks @lamenezes)
- 1.9.0 Add support for boto3 (thanks @desdm, @foorbarna). Fix deepcopy issue for response headers when *decode_compressed_response* is enabled (thanks @nickdirienzo)
- 1.8.0 Fix for Serialization errors with JSON adapter (thanks @aliaksandrb). Avoid concatenating bytes with strings (thanks @jaysonsantos). Exclude __pycache__ dirs & compiled files in sdist (thanks @koobs). Fix Tornado support behavior for Tornado 3 (thanks @abhinav). decode_compressed_response option and filter (thanks @jayvdb).
- 1.7.4 [#217] Make use_cassette decorated functions actually return a value (thanks @bcen). [#199] Fix path transfromation defaults. Better headers dictionary management.
- 1.7.3 [#188] `additional_matchers` kwarg on `use_cassette`. [#191] Actually support passing multiple before_record_request functions (thanks @agriffis).
- 1.7.2 [#186] Get effective_url in tornado (thanks @mvschaik), [#187] Set request_time on Response object in tornado (thanks @abhinav).
- 1.7.1 [#183] Patch `fetch_impl` instead of the entire HTTPClient class for Tornado (thanks @abhinav).
- 1.7.0 [#177] Properly support coroutine/generator decoration. [#178] Support distribute (thanks @graingert). [#163] Make compatibility between python2 and python3 recorded cassettes more robust (thanks @gward).
- 1.6.1 [#169] Support conditional requirements in old versions of pip, Fix RST parse errors generated by pandoc, [Tornado] Fix unsupported features exception not being raised, [#166] content-aware body matcher.
- 1.6.0 [#120] Tornado support (thanks @abhinav), [#147] packaging fixes (thanks @graingert), [#158] allow filtering post params in requests (thanks @MrJohz), [#140] add xmlrpclib support (thanks @Diaoul).
- 1.5.2 Fix crash when cassette path contains cassette library directory (thanks @gazpachoking).
- 1.5.0 Automatic cassette naming and 'application/json' post data filtering (thanks @marco-santamaria).
- 1.4.2 Fix a bug caused by requests 2.7 and chunked transfer encoding
- 1.4.1 Include README, tests, LICENSE in package. Thanks @ralphbean.
- 1.4.0 Filter post data parameters (thanks @eadmundo), support for posting files through requests, inject_cassette kwarg to access cassette from `use_cassette` decorated function, `with_current_defaults` actually works (thanks @samstav).
- 1.3.0 Fix/add support for urllib3 (thanks @aisch), fix default port for https (thanks @abhinav).
- 1.2.0 Add custom_patches argument to VCR/Cassette objects to allow users to stub custom classes when cassettes become active.
- 1.1.4 Add force reset around calls to actual connection from stubs, to ensure compatibility with the version of httplib/urlib2 in python 2.7.9.
- 1.1.3 Fix python3 headers field (thanks @rtaboada), fix boto test (thanks @telaviv), fix new_episodes record mode (thanks @jashugan), fix Windows connectionpool stub bug (thanks @gazpachoking), add support for requests 2.5

- 1.1.2 Add urllib==1.7.1 support. Make json serialize error handling correct Improve logging of match failures.
- 1.1.1 Use function signature preserving `wrapt.decorator` to write the decorator version of use_cassette in order to ensure compatibility with py.test fixtures and python 2. Move all request filtering into the `before_record_callable`.
- 1.1.0 Add `before_record_response`. Fix several bugs related to the context management of cassettes.
- 1.0.3: Fix an issue with requests 2.4 and make sure case sensitivity is consistent across python versions
- 1.0.2: Fix an issue with requests 2.3
- 1.0.1: Fix a bug with the new ignore requests feature and the once record mode
- 1.0.0: *BACKWARDS INCOMPATIBLE*: Please see the 'upgrade' section in the README. Take a look at the matcher section as well, you might want to update your `match_on` settings. Add support for filtering sensitive data from requests, matching query strings after the order changes and improving the built-in matchers, (thanks to @mshytikov), support for ignoring requests to certain hosts, bump supported Python3 version to 3.4, fix some bugs with Boto support (thanks @marusich), fix error with URL field capitalization in README (thanks @simon-weber), added some log messages to help with debugging, added `all_played` property on cassette (thanks @mshytikov)
- 0.7.0: VCR.py now supports Python 3! (thanks @asundg) Also I refactored the stub connections quite a bit to add support for the putrequest and putheader calls. This version also adds support for httplib2 (thanks @nilp0inter). I have added a couple tests for boto since it is an http client in its own right. Finally, this version includes a fix for a bug where requests wasn't being patched properly (thanks @msabramo).
- 0.6.0: Store response headers as a list since a HTTP response can have the same header twice (happens with set-cookie sometimes). This has the added benefit of preserving the order of headers. Thanks @smallcode for the bug report leading to this change. I have made an effort to ensure backwards compatibility with the old cassettes' header storage mechanism, but if you want to upgrade to the new header storage, you should delete your cassettes and re-record them. Also this release adds better error messages (thanks @msabramo) and adds support for using VCR as a decorator (thanks @smallcode for the motivation)
- 0.5.0: Change the `response_of` method to `responses_of` since cassettes can now contain more than one response for a request. Since this changes the API, I'm bumping the version. Also includes 2 bugfixes: a better error message when attempting to overwrite a cassette file, and a fix for a bug with requests sessions (thanks @msabramo)
- 0.4.0: Change default request recording behavior for multiple requests. If you make the same request multiple times to the same URL, the response might be different each time (maybe the response has a timestamp in it or something), so this will make the same request multiple times and save them all. Then, when you are replaying the cassette, the responses will be played back in the same order in which they were received. If you were making multiple requests to the same URL in a cassette before version 0.4.0, you might need to regenerate your cassette files. Also, removes support for the cassette.play_count counter API, since individual requests aren't unique anymore. A cassette might contain the same request several times. Also removes secure overwrite feature since that was breaking overwriting files in Windows, and fixes a bug preventing request's automatic body decompression from working.
- 0.3.5: Fix compatibility with requests 2.x
- 0.3.4: Bugfix: close file before renaming it. This fixes an issue on Windows. Thanks @smallcode for the fix.
- 0.3.3: Bugfix for error message when an unreigstered custom matcher was used
- 0.3.2: Fix issue with new config syntax and the `match_on` parameter. Thanks, @chromy!
- 0.3.1: Fix issue causing full paths to be sent on the HTTP request line.
- 0.3.0: *Backwards incompatible release* - Added support for record modes, and changed the default recording behavior to the "once" record mode. Please see the documentation on record modes for more. Added support for custom request matching, and changed the default request matching behavior to match only on the URL

and method. Also, improved the httplib mocking to add support for the `HTTPConnection.send()` method. This means that requests won't actually be sent until the response is read, since I need to record the entire request in order to match up the appropriate response. I don't think this should cause any issues unless you are sending requests without ever loading the response (which none of the standard httplib wrappers do, as far as I know. Thanks to @fatuhoku for some of the ideas and the motivation behind this release.

- 0.2.1: Fixed missing modules in setup.py
- 0.2.0: Added configuration API, which lets you configure some settings on VCR (see the README). Also, VCR no longer saves cassettes if they haven't changed at all and supports JSON as well as YAML (thanks @sirpengi). Added amazing new skeumorphic logo, thanks @hairarrow.
- 0.1.0: *backwards incompatible release - delete your old cassette files*: This release adds the ability to access the cassette to make assertions on it, as well as a major code refactor thanks to @dlecocq. It also fixes a couple longstanding bugs with redirects and HTTPS. [#3 and #4]
- 0.0.4: If you have libyaml installed, vcrpy will use the c bindings instead. Speed up your tests! Thanks @dlecocq
- 0.0.3: Add support for requests 1.2.3. Support for older versions of requests dropped (thanks @vitormazzi and @bryanhelmig)
- 0.0.2: Add support for requests / urllib3
- 0.0.1: Initial Release
- genindex
- modindex
- search

Python Module Index

V

vcr.cassette, 17
vcr.config, 17
vcr.filters, 18
vcr.matchers, 18
vcr.patch, 19
vcr.request, 19
vcr.serialize, 19

Index

Symbols

`__init__()` (*vcr.cassette.Cassette method*), 17
`__init__()` (*vcr.cassette.CassetteContextDecorator method*), 18
`__init__()` (*vcr.request.Request method*), 19

A

`all_played` (*vcr.cassette.Cassette attribute*), 17
`append()` (*vcr.cassette.Cassette method*), 17

C

`Cassette` (*class in vcr.cassette*), 17
`CASSETTE_FORMAT_VERSION` (*in module vcr.serialize*), 19
`CassetteContextDecorator` (*class in vcr.cassette*), 17

D

`decode_response()` (*in module vcr.filters*), 18

F

`find_requests_with_most_matches()` (*vcr.cassette.Cassette method*), 17

G

`get_assertion_message()` (*in module vcr.matchers*), 18
`get_matchers_results()` (*in module vcr.matchers*), 18

H

`HeadersDict` (*class in vcr.request*), 19

L

`load()` (*vcr.cassette.Cassette class method*), 17

P

`play_response()` (*vcr.cassette.Cassette method*), 17

R

`remove_headers()` (*in module vcr.filters*), 18
`remove_post_data_parameters()` (*in module vcr.filters*), 18
`remove_query_parameters()` (*in module vcr.filters*), 18
`replace_headers()` (*in module vcr.filters*), 18
`replace_post_data_parameters()` (*in module vcr.filters*), 18
`replace_query_parameters()` (*in module vcr.filters*), 18
`Request` (*class in vcr.request*), 19
`responses_of()` (*vcr.cassette.Cassette method*), 17

V

`vcr.cassette` (*module*), 17
`vcr.config` (*module*), 17
`vcr.filters` (*module*), 18
`vcr.matchers` (*module*), 18
`vcr.patch` (*module*), 19
`vcr.request` (*module*), 19
`vcr.serialize` (*module*), 19