WalkScore API

Release 1.0.1

Insight Industry Inc.

Jan 04, 2020

CONTENTS:

(Unofficial) Python Bindings for the WalkScore API

Version Compatability

The **WalkScore Library** is designed to be compatible with:

- Python 3.6 or higher

Branch	Unit Tests
latest	
v.1.0	
develop	

CHAPTER

ONE

QUICKSTART: PATTERNS AND BEST PRACTICES

1.1 Installation

To install **WalkScore**, just execute:

```
$ pip install walkscore-api
```

1.2 Initializing the API

To initialize the `WalkScoreAPI` object all you need to do is instantiate it:

```
from walkscore import WalkScoreAPI

# supplying an API key
walkscore = WalkScoreAPI(api_key = 'MY API KEY GOES HERE')

# using an API key in the "WALKSCORE_API_KEY" environment variables
walkscore = WalkScoreAPI()
```

1.3 Configuring the HTTP Client

You can heavily customize the HTTP client used by the WalkScore Library. By default, the library will look for HTTP libraries in the following order:

- urlfetch
- requests
- pycurl
- urllib (Python standard library)

Tip: You can also override the HTTP client by subclassing the *HTTPClient* class.

There are three ways to customize / configure the HTTP client:

1. Subclass the *HTTPClient* class.
2. Supply a proxy URL.
3. Configure the maximum number of retries.

1.3.1 Subclassing the Client

```
from walkscore import WalkScoreAPI

from my_custom_client import MyCustomHTTPClient

walkscore = WalkScoreAPI(http_client = MyCustomHTTPClient)
```

1.3.2 Configuring a Proxy

```
from walkscore import WalkScoreAPI

walkscore = WalkScoreAPI(proxy = 'http://www.some-proxy-url')
```

1.3.3 Configuring the Maximum Number of Retries

If the WalkScore Library is unable to get a response from the WalkScore API, it will automatically apply an exponential backoff/retry strategy. However, you can configure the maximum number of retries that it attempts. This can be configured in two ways:

1. By setting the `BACKOFF_DEFAULT_TRIES` environment variable.
2. By passing the maximum number of retries in the `max_retries` argument:

   ```
   from walkscore import WalkScoreAPI

   walkscore = WalkScoreAPI(max_retries = 5)
   ```

1.4 Getting Scores

To retrieve scores, all you need to do is to call the `get_score()` method on the initialized API:

```
from walkscore import WalkScoreAPI

walkscore = WalkScoreAPI(api_key = 'MY API KEY GOES HERE')

result = walkscore.get_score(latitude = 123.45, longitude = 54.321)
```

Note: In order to retrieve a score from the API, you *must* supply the latitude and longitude of the point you are looking for. The WalkScore API does *not* support geocoding based on addresses, although an address can provide more precise results if you supply it as well.

Tip: In order to get better performance out of the underlying WalkScore API, you may want to suppress the calculation / retrieval of *TransitScores* and/or *BikeScores* if you don't need them. To do that, all you need to do is pass the appropriate arguments into the `get_score()` method:

```
result = walkscore.get_score(latitude = 123.45,
                             longitude = 54.321,
                             return_transit_score = False,
                             return_bike_score = False)
```

The results returned by the `get_score()` method are always `LocationScore` instances.

1.5 Working with Scores

When the WalkScore Library has retrieved a score for a given set of coordinates, you can work with it as any other Python object. See the `LocationScore` reference documentation for more insight into its properties.

However, there are a number of key serialization / deserialization methods that you may find useful:

- `.to_json()` which returns a JSON representation of the location score, either normalized to a cleaner/more consistent structure preferred by the WalkScore Library or mirroring the WalkScore API's JSON structure
- `.from_json()` which returns a `LocationScore` instance generated from a JSON string
- `.to_dict()` which returns a `dict` representation fo the location score, either normalized to a cleaner/more consistent structure preferred by the WalkScore Library or mirroring the WalkScore API's JSON structure
- `.from_dict()` which returns a `LocationScore` instance generated from a `dict`

CHAPTER TWO

API REFERENCE

2.1 WalkScoreAPI

class WalkScoreAPI(*api_key=None*, *http_client=None*, *proxy=None*, *max_retries=None*)

The Python object which exposes the WalkScore API's functionality.

Parameters

- **api_key** (`str` / `None`) – The API key provided by WalkScore used to authenticate your application. If `None` or not specified will default to the `WALKSCORE_API_KEY` environment variable if present, and `None` if not.
- **http_client** (*`HTTPClient`*) – The HTTP client instance to use for the execution of requests. If not overridden, will default to urlfetch, requests, pycurl, urllib2 in order based on whether they are available in the environment.

  Tip: You can override the HTTP client by supplying a *`HTTPClient`* instance to the method.

- **proxy** (`str` / `None`) – The URL to use as an HTTP proxy. Defaults to `None`.
- **max_retries** (`int`) – Determines the maximum number of HTTP request attempts to make on network failure before giving up. If not specified, defaults to environment variable `BACKOFF_DEFAULT_TRIES` or `3` if not available.

get_score(*latitude*, *longitude*, *address=None*, *return_transit_score=True*, *return_bike_score=True*, *max_retries=None*)

Retrieve the *WalkScore*, *TransitScore*, and/or *BikeScore* for a given location from the WalkScore API.

Parameters

- **latitude** (*`numeric`*) – The latitude of the location whose score(s) should be retrieved.
- **longitude** (*`numeric`*) – The longitude of the location whose score(s) should be retrieved.

- **address** (`str` / `None`) – The address whose score(s) should be retrieved. Defaults to `None`.
- **return_transit_score** (`bool`) – If `True`, will return the location's *TransitScore*. Defaults to `True`.
- **return_bike_score** (`bool`) – If `True`, will return the location's *BikeScore*. Defaults to `True`.
- **max_retries** (`None` / `int`) – The maximum number of retries to attempt if the WalkScore API times out or otherwise fails to return a response. If `None`, will apply the default the configured when initializing the WalkScore API object. To suppress all retries, set to 0. Defaults to `None`.

Returns The location's *WalkScore*, *TransitScore*, and *BikeScore* with meta-data.

Return type *LocationScore*

Raises

- ***AuthenticationError*** – if the API key is invalid
- ***ScoreInProgressError*** – if the score is being calculated and is not currently available
- ***WalkScoreError*** – if an internal WalkScore API error occurred
- ***QuotaError*** – if your daily quota has been exceeded
- ***BlockedIPError*** – if your IP address has been blocked
- ***InvalidCoordinatesError*** – if your latitude/longitude coordinates are not valid

property api_key

The API key used to sign requests made against the API.

Return type `str` / `None`

property http_client

The object instance to use as the HTTP client to make HTTP requests against the WalkScore API.

Return type *HTTPClient*

property max_retries

The number of attempts to make on network connectivity-related API failures.

Return type `int`

property proxy

The URL to use as a proxy for requests made to the WalkScore API.

Return type `str` / `None`

2.2 LocationScore

class LocationScore(*address=None, original_latitude=None, original_longitude=None, status=None, walk_score=None, walk_description=None, walk_updated=None, transit_score=None, transit_description=None, transit_summary=None, bike_score=None, bike_description=None, logo_url=None, more_info_icon=None, more_info_link=None, help_link=None, snapped_latitude=None, snapped_longitude=None, property_page_link=None*)

Object representation of a location's scoring data returned from the WalkScore API.

Parameters

- **address** (`str` / `None`) – The address originally supplied to the WalkScore API. Defaults to `None`.
- **original_latitude** (numeric / `None`) – The latitude value originally supplied to the WalkScore API. Defaults to `None`.
- **original_longitude** (numeric / `None`) – The longitude value originally supplied to the WalkScore API. Defaults to `None`.
- **status** (`int` / `None`) – The status returned from the WalkScore API. Defaults to `None`.
- **walk_score** (`int` / `None`) – The *WalkScore* for the location. Deafults to `None`
- **walk_description** (`str` / `None`) – An English characterization of the *WalkScore*, e.g. "Somewhat Walkable". Defaults to `None`.
- **walk_updated** (`datetime` / `None`) – The timestamp when the *WalkScore* was calculated. Defaults to `None`
- **transit_score** (`int` / `None`) – The *TransitScore* for the location. Deafults to `None`
- **transit_description** (`str` / `None`) – An English characterization of the *TransitScore*, e.g. "Rider's Paradise". Defaults to `None`.
- **transit_summary** (`str` / `None`) – Notes on the transit options accessible from the location. Defaults to `None`.
- **bike_score** (`int` / `None`) – The *BikeScore* for the location. Deafults to `None`
- **bike_description** (`str` / `None`) – An English characterization of the *BikeScore*, e.g. "Very Bikeable". Defaults to `None`.
- **logo_url** (`str` / `None`) – The URL of the WalkScore logo. Defaults to `None`.
- **more_info_icon** (`str` / `None`) – The URL to the icon to use when linking to more information. Defaults to `None`.
- **more_info_link** (`str` / `None`) – The URL to link to when providing more information. Defaults to `None`.
- **help_link** (`str` / `None`) – A link to the "How Walk Score Works" page. Defaults to `None`.
- **snapped_latitude** (numeric / `None`) – The latitude for the location, snapped to a grid of approximately 500 ft. by 500 ft. Defaults to `None`.
- **snapped_longitude** (numeric / `None`) – The longitude for the location, snapped to a grid of approximately 500 ft. by 500 ft. Defaults to `None`.
- **property_page_link** (`str` / `None`) – The URL to the walkscore.com map and score for the location. Defaults to `None`.

classmethod from_dict(*obj*, *api_compatible=False*)
Create a `LocationScore` instance from a `dict` representation.

Parameters

- **obj** (`dict`) – The `dict` representation of the location score.
- **api_compatible** (`bool`) – If `True`, expects `obj` to be a `dict` whose structure is compatible with the JSON object returned by the WalkScore API. If `False`, expects a slightly more normalized `dict` representation. Defaults to `False`.

Returns `LocationScore` representation of `obj`.

Return type `LocationScore`

classmethod from_json(*obj*, *api_compatible=False*)
Create a `LocationScore` instance from a JSON representation.

Parameters

- **obj** (`str` or `bytes`) – The JSON representation of the location score.
- **api_compatible** (`bool`) – If `True`, expects `obj` to be a JSON object whose structure is compatible with the JSON object returned by the WalkScore API. If `False`, expects a slightly more normalized representation. Defaults to `False`.

Returns `LocationScore` representation of `obj`.

Return type `LocationScore`

to_dict(*api_compatible=False*)
Serialize the `LocationScore` to a `dict`.

Parameters **api_compatible** (`bool`) – If `True`, returns a `dict` whose structure is compatible with the JSON object returned by the WalkScore API. If `False`, returns a slightly more normalized `dict` representation. Defaults to `False`.

Returns `dict` representation of the object

Return type `dict`

to_json(*api_compatible=False*)
Serialize the `LocationScore` to a *JSON* string.

Parameters **api_compatible** (`bool`) – If `True`, returns a JSON object whose structure is compatible with the JSON object returned by the WalkScore API. If `False`, returns a slightly more normalized structure. Defaults to `False`.

Returns `str` representation of a JSON object

Return type `str`

property address
The original address supplied for the `LocationScore`.

Return type `str`

property bike_description
A textual description of the location's bike-ability.

Return type `str`

property bike_score
The *TransitScore* for the location, measuring bike-ability on a scale from 0 to 100.

Return type `int`

property help_link
URL to the "How WalkScore Works" page.

Return type `str`

property logo_url
URL to the WalkScore logo.

Return type `str`

property more_info_icon
URL to the question mark icon to display next to the Score.

Return type `str`

property more_info_link
URL for the question mark displayed next to the Score to link to.

Return type `str`

property original_coordinates
The coordinates of the location as originally supplied.

Return type `tuple` of longitude and latitude as `float` values

property original_latitude
The latitude of the location as originally supplied.

Return type `float`

property original_longitude
The longitude of the location as originally supplied.

Return type `float`

property property_page_link
URL to the walkscore.com score and map for the location.

Return type `str`

property snapped_coordinates
The coordinates of the location as returned by the API.

Return type `tuple` of longitude and latitude as `float` values

property snapped_latitude
The latitude of the location as returned by the API.

Return type `float`

property snapped_longitude
The longitude of the location as returned by the API.

Return type `float`

property status
Status Code of the result.

Return type `int`

property transit_description
A textual description of the location's ease-of-transit.

Return type `str`

property transit_score
The *TransitScore* for the location, measuring ease-of-transit on a scale from 0 to 100.

Return type `int`

property transit_summary
A textual summary of the location's ease-of-transit.

Return type `str`

property walk_description
A textual description of the location's walkability.

Return type `str`

property walk_score
The *WalkScore* for the location, measuring walkability on a scale from 0 to 100.

Return type `int`

property walk_updated
The timestamp for when the location's *WalkScore* was last updated.

Return type `datetime`

2.3 HTTPClient

class HTTPClient(*verify_ssl_certs=True*, *proxy=None*)
Base class that provides HTTP connectivity.

close()
Closes an existing HTTP connection/session.

request(*method*, *url*, *parameters=None*, *headers=None*, *request_body=None*)
Execute a standard HTTP request.

Parameters

- **method** (`str`) – The HTTP method to use for the request. Accepts *GET*, *HEAD*, *POST*, *PATCH*, *PUT*, or *DELETE*.
- **url** (`str`) – The URL to execute the request against.
- **parameters** (`dict` / `None`) – URL parameters to submit with the request. Defaults to `None`.
- **headers** (`dict` / `None`) – HTTP headers to submit with the request. Defaults to `None`.
- **request_body** (`None` / `dict` / `str` / `bytes`) – The data to supply in the body of the request. Defaults to `None`.

Returns The content of the HTTP response, the status code of the HTTP response, and the headers of the HTTP response.

Return type `tuple` of `bytes`, `int`, and `dict`

Raises

- **`ValueError`** – if `method` is not either `GET`, `HEAD`, `POST`, `PATCH`, `PUT` or `DELETE`
- **`ValueError`** – if `url` is not a valid URL
- **`ValueError`** – if `headers` is not empty and is not a `dict`
- ***`HTTPTimeoutError`*** – if the request times out

- ***SSLError*** – if the request fails SSL certificate verification
- ***WalkScoreError*** – *or sub-classes* for other errors returned by the API

request_with_retries (*method*, *url*, *parameters=None*, *headers=None*, *request_body=None*)

Execute a standard HTTP request with automatic retries on failure.

Parameters

- **method** (`str`) – The HTTP method to use for the request. Accepts *GET*, *HEAD*, *POST*, *PATCH*, *PUT*, or *DELETE*.
- **url** (`str`) – The URL to execute the request against.
- **parameters** (`dict` / `None`) – URL parameters to submit with the request. Defaults to `None`.
- **headers** (`dict` / `None`) – HTTP headers to submit with the request. Defaults to `None`.
- **request_body** (`None` / `dict` / `str` / `bytes`) – The data to supply in the body of the request. Defaults to `None`.

Note: This method will apply an exponential backoff strategy to retry the API request if it times out. By default:

- requests that can be retried will be retried up to `3` times, but this can be overridden by setting a `BACKOFF_DEFAULT_TRIES` environment variable with the maximum number of attempts to make
- there is no maximum delay to wait before final failure, but this can be overridden by setting a `BACKOFF_DEFAULT_DELAY` environment variable with the maximum number of seconds to wait (across all attempts) before failing.

Raises

- **ValueError** – if `method` is not either `GET`, `HEAD`, `POST`, `PATCH`, `PUT` or `DELETE`
- **ValueError** – if `url` is not a valid URL
- ***HTTPTimeoutError*** – if the request times out after repeated attempts
- ***SSLError*** – if the request fails SSL certificate verification
- ***WalkScoreError*** – *or sub-classes* for other errors returned by the API

CHAPTER

THREE

ERROR REFERENCE

3.1 Handling Errors

3.1.1 Stack Traces

Because **WalkScore** produces exceptions which inherit from the standard library, it leverages the same API for handling stack trace information. This means that it will be handled just like a normal exception in unit test frameworks, logging solutions, and other tools that might need that information.

3.2 WalkScore Errors

3.2.1 WalkScoreError (from `ValueError`)

`class WalkScoreError`

Base error raised by **WalkScore**. Inherits from `ValueError`.

3.2.2 AuthenticationError (from `WalkScoreError`)

`class AuthenticationError`

Error raised when attempting to retrieve a score with an invalid API key.

3.2.3 InternalAPIError (from `WalkScoreError`)

`class InternalAPIError`

Internal error within the WalkScore API itself. Inherits from *`WalkScoreError`*.

3.2.4 BlockedIPError (from `WalkScoreError`)

`class BlockedIPError`

Error raised when attempting to retrieve a score from a blocked IP address.

3.2.5 QuotaError (from `WalkScoreError`)

`class QuotaError`

Error raised when you have exceeded your daily quota.

3.2.6 ScoreInProgressError (from `WalkScoreError`)

`class ScoreInProgressError`

Error raised when a score for the location supplied is being calculated and is not yet available.

3.2.7 InvalidCoordinatesError (from `WalkScoreError`)

`class InvalidCoordinatesError`

Error raised when the coordinates supplied for a location are invlaid.

3.2.8 BindingError (from `WalkScoreError`)

`class BindingError`

Error produced when the WalkScore Library has an incorrect API binding.

3.2.9 HTTPConnectionError (from `WalkScoreError`)

`class HTTPConnectionError`

Error produced when the WalkScore Library is unable to connect to the API, but did not time out.

3.2.10 HTTPTimeoutError (from `HTTPConnectionError`)

`class HTTPTimeoutError`

Error produced when the API times out or returns a `Status Code:  504`.

This error indicates that the underlying API timed out and did not return a result.

3.2.11 SSLError (from `WalkScoreError`)

`class SSLError`

Error produced when an SSL certificate cannot be verified, returns a `Status Code:  495`.

CHAPTER

FOUR

CONTRIBUTING TO WALKSCORE

Note: As a general rule of thumb, the **WalkScore** library applies **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:

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

4.1 Design Philosophy

WalkScore 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.

4.2 Style Guide

4.2.1 Basic Conventions

- Do not terminate lines with semicolons.
- 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()
```

(continues on next page)

(continued from previous page)

```
# 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.

4.2.2 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`.

4.2.3 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,
                     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).

4.2.4 Documentation Conventions

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

Sphinx[1]

Sphinx[1] is used to organize the library's documentation into this lovely readable format (which is also 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`.
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**.

[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

4.3 Dependencies

- Validator-Collection v1.3.0 or higher
- Backoff-Utils v.1.0.0 or higher

4.4 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:

   ```
   walkscore/ $ pip install -r requirements.txt
   ```

And you should be good to go!

4.5 Ideas and Feature Requests

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

4.6 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 **WalkScore** testing approach please see: *Testing WalkScore*

4.7 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.

4.8 Building Documentation

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

```
walkscore-api/ $ cd docs
walkscore-api/docs $ make html
```

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

```
walkscore/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.

4.9 References

CHAPTER

FIVE

TESTING WALKSCORE

Contents

5.1 Testing Philosophy

Note: Unit tests for **WalkScore** 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 **WalkScore**, 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, there are certain elements of code that are repeated in almost all test functions, as doing so will make future readability and maintenance of the test suite easier.
- **Coverage matters. . . kind of.** We have documented the primary intended behavior of every function in the **WalkScore** library, and the most-likely failure modes that can be expected. At the time of writing, we have about 85% code coverage. Yes, yes: We know that is less than 100%. But there are edge cases which are

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

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.

5.2 Test Organization

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

- `test_locationscore.py` tests the LocationScore class found in `walkscore/locationscore.py`

Certain test modules are 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`.

5.3 Configuring & Running Tests

5.3.1 Installing with the Test Suite

Installing via pip

```
$ pip install walkscore-api[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.

5.3.2 Command-line Options

WalkScore does not use any custom command-line options in its test suite.

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

```
walkscore-api $ cd tests/
walkscore-api/tests/ $ pytest --help
```

5.3.3 Configuration File

Because **WalkScore** has a very simple test suite, we have not prepared a `pytest.ini` configuration file.

5.3.4 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'
```

5.4 Skipping Tests

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

5.5 Incremental Tests

Note: The **WalkScore** test suite does support incremental testing, 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
        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

SIX

RELEASE HISTORY

Contents

6.1 Release v.1.0.1

- Fixed false negative in unit tests.
- Fixed TravisCI configuration in Python 3.6 / PyCURL environment.
- Added Python 3.8 to test matrix.

6.2 Release v.1.0.0

- First public release

CHAPTER

SEVEN

GLOSSARY

WalkScore Measures walkability on a scale from 0 - 100 based on walking routes to destinations such as grocery stores, schools, parks, restaurants, and retail.

TransitScore Measures transit accessibility on a scale from 0 - 100. Calculates distance to closest stop on each route, analyzes route frequency and type.

BikeScore Measures bike accessibility on a scale from 0 - 100 based on bike infrastructure, topography, destinations and road connectivity.

JSON A lightweight data-interchange format that has become the *de facto* standard for communication across internet-enabled APIs.

For a formal definition, please see the ECMA-404 Standard: JSON Data Interchange Syntax

WALKSCORE API LICENSE

The **WalkScore Library** is a Python library that provides Python bindings for the WalkScore API. It enables you to retrieve *WalkScores*, *TransitScores*, and *BikeScores* from the API within your Python code.

Warning: The **WalkScore Library** is completely unaffiliated with WalkScore. It is entirely unofficial and was developed based on publicly available documentation of the WalkScore APIs published to the WalkScore website. Use of WalkScore is subject to WalkScore's licenses and terms of service, and this library is not endorsed by WalkScore or any affiliates thereof.

Contents

CHAPTER NINE

INSTALLATION

To install **WalkScore**, just execute:

```
$ pip install walkscore-api
```

9.1 Dependencies

- Validator-Collection v1.3.0 or higher
- Backoff-Utils v.1.0.0 or higher

9.2 Key WalkScore Features

- Python representation of *WalkScores*, *TransitScores*, and *BikeScores*
- Easy serialization and deserialization of API responses to Python objects, `dict` objects or *JSON*
- Built-in back-off/retry logic if the WalkScore API is unstable at any moment in time
- Robust error handling to surface meaningful information to help you debug your code.

CHAPTER

TEN

HELLO, WORLD AND BASIC USAGE

10.1 1. Import the WalkScore API

```
from walkscore import WalkScoreAPI
```

10.2 2. Initialize the API

You can either use a single object to communicate with all of the available WalkScore APIs, or initialize a single object for each API:

```
api_key = 'YOUR API KEY GOES HERE'

score_api = WalkScoreAPI(api_key = api_key)
```

10.3 3. Retrieve a Score

```
address = '123 Anyplace St Anywhere, AK 12345'

result = score_api.get_score(longitude = 123.45, latitude = 54.321, address = address)

# the WalkScore for the location
result.walk_score

# the TransitScore for the location
result.transit_score

# the BikeScore for the location
result.bike_score
```

CHAPTER ELEVEN

QUESTIONS AND ISSUES

You can ask questions and report issues on the project's Github Issues Page

CHAPTER

TWELVE

CONTRIBUTING

We welcome contributions and pull requests! For more information, please see the *Contributor Guide*

CHAPTER THIRTEEN

TESTING

We use TravisCI for our build automation and ReadTheDocs for our documentation.

Detailed information about our test suite and how to run tests locally can be found in our *Testing Reference*.

CHAPTER FOURTEEN

LICENSE

WalkScore is made available under an *MIT License*.

CHAPTER FIFTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX