timezonefinder

Release /bin/sh: 1: poetry: not found

Jannik Michelfeit

Mar 14, 2024

CONTENTS:

a fast and lightweight python package for looking up the corresponding timezone for given coordinates on earth entirely offline.

CHAPTER

ONE

GETTING STARTED

1.1 Installation

```
pip install timezonefinder
```

in case you are using `pytz`, also require it via its extra to avoid incompatibilities (e.g. due to updated timezone names):

```
pip install timezonefinder[pytz]
```

for improved speed also install the optional dependency `numba` via its extra (also check the *performance chapter*):

```
pip install timezonefinder[numba]
```

For installation within a Conda environment see instructions at conda-forge feedstock

1.2 Dependencies

`python3.8+`, `numpy`, `h3`, `cffi`

optional: `numba`

(cf. `pyproject.toml`)

1.3 Basic Usage

```
from timezonefinder import TimezoneFinder

tf = TimezoneFinder()  # reuse

query_points = [(13.358, 52.5061), ...]
for lng, lat in query_points:
    tz = tf.timezone_at(lng=lng, lat=lat)  # 'Europe/Berlin'
```

All available features of this package are explained *HERE*.

Examples for common use cases can be found *HERE*.

CHAPTER

TWO

USAGE

Note: Also check out the *API documentation* or the code.

2.1 Initialisation

Create a new instance of the *TimezoneFinder class* to be reused for multiple consequent timezone queries:

```
from timezonefinder import TimezoneFinder

tf = TimezoneFinder()  # reuse
```

Use the argument `bin_file_location` to use data files from another location (e.g. *your own compiled files*):

```
tf = TimezoneFinder(bin_file_location="path/to/files")
```

2.2 timezone_at()

This is the default function to check which timezone a point lies within. If no timezone has been matched, `None` is being returned.

```
tz = tf.timezone_at(lng=13.358, lat=52.5061)  # 'Europe/Berlin'
tz = tf.timezone_at(lng=1.0, lat=50.5)  # 'Etc/GMT'
```

Note: To reduce the risk of mixing up the coordinates, the arguments `lng` and `lat` have to be given as keyword arguments

Note: This function is optimized for speed: The last possible timezone in proximity is always returned (without checking if the point is really included).

2.3 timezone_at_land()

This package includes ocean timezones (`Etc/GMT...`). If you want to explicitly receive only "land" timezones use

```
tz = tf.timezone_at_land(lng=13.358, lat=52.5061)  # 'Europe/Berlin'
tz = tf.timezone_at_land(lng=1.0, lat=50.5)  # None
```

2.4 unique_timezone_at()

For fast execution `timezonefinder` internally uses precomputed "shortcuts" which store the possible zones in proximity. Call `unique_timezone_at()` if you want to compute an exact result without actually performing "point-in-polygon" tests (<- computationally expensive). This function will return `None` when the correct zone cannot be uniquely determined without further computation.

```
tf.unique_timezone_at(lng=longitude, lat=latitude)
```

Note: The "lightweight" class *TimezoneFinderL*, which is using only shortcuts, also supports just querying the most probable timezone.

2.5 certain_timezone_at()

Note: DEPRECATED: Due to the included ocean timezones one zone will always be matched. Use `timezone_at()` or `timezone_at_land()` instead.

This function is for making sure a point is really inside a timezone. It is slower, because all polygons (with shortcuts in that area) are being checked until one polygon is matched. `None` is being returned in the case of no match.

```
tz = tf.certain_timezone_at(lng=13.358, lat=52.5061)  # 'Europe/Berlin'
```

Note: Due to the "point-in-polygon-test" algorithm being used, the state of a point on the edge of a (timezone) polygon is undefined. For those kind of points the return values is hence uncertain and might be `None`. This applies for example for all points with lng=+-180.0, because the timezone polygons in the data set are being cropped at the 180 longitude border.

2.6 closest_timezone_at()

removed in version `6.0.0`

2.7 get_geometry()

For querying a timezone for its geometric multi-polygon shape use `get_geometry()`. output format: `[ [polygon1, hole1,...), [polygon2, ...], ...]` and each polygon and hole is itself formated like: `([longitudes], [latitudes])` or `[(lng1,lat1), (lng2,lat2),...]` if `coords_as_pairs=True`.

```
tf.get_geometry(tz_name="Africa/Addis_Ababa", coords_as_pairs=True)
tf.get_geometry(tz_id=400, use_id=True)
```

2.8 TimezoneFinderL

TimezoneFinderL is a light version of the *TimezoneFinder class*. It is useful for quickly suggesting probable timezones without using as many computational resources (cf. *speed tests*). Instead of using timezone polygon data this class instantly returns the timezone just based on precomputed "shortcuts".

Check the (*API documentation*) of `TimezoneFinderL`.

The most probable zone in proximity can be retrieved with `timezone_at()`:

```
from timezonefinder import TimezoneFinderL

tf = TimezoneFinderL(in_memory=True)  # reuse

query_points = [(13.358, 52.5061), ...]
for lng, lat in query_points:
    tz = tf.timezone_at(lng=lng, lat=lat)  # 'Europe/Berlin'
```

Certain results can be retrieved with `unique_timezone_at()`:

```
tf.unique_timezone_at(lng=13.358, lat=52.5061)  # 'Europe/Berlin'
```

Note: If you only use `TimezoneFinderL`, you may delete all data files except `timezone_names.json`, `shortcuts.bin` to obtain a truly lightweight installation.

2.9 Using vectorized input

Check numpy.vectorize and pandas.DataFrame.apply

2.10 Calling timezonefinder from the command line

A command line script is being installed as part of this package.

Command Line Syntax:

```
timezonefinder [-h] [-v] [-f {0,1,2,3,4,5}] lng lat
```

Example:

```
timezonefinder -f 4 40.5 11.7
```

With `-v` you get verbose output, without it only the timezone name is being printed. With the argument of the flag `-f` one can choose between the different functions to be called:

```
0: TimezoneFinder.timezone_at() = default
1: TimezoneFinder.certain_timezone_at()
2: removed
3: TimezoneFinderL.timezone_at()
4: TimezoneFinderL.timezone_at_land()
5: TimezoneFinder.timezone_at_land()
```

Note: This will be orders of magnitude slower than using the package directly from within python as a separate Timezonefinder() instance is being

CHAPTER

THREE

USE CASES:

3.1 Creating aware datetime objects

```
# first pip install pytz
from pytz import timezone, utc
from pytz.exceptions import UnknownTimeZoneError

# tzinfo has to be None (means naive)
naive_datetime = YOUR_NAIVE_DATETIME

try:
    tz = timezone(timezone_name)
    aware_datetime = naive_datetime.replace(tzinfo=tz)
    aware_datetime_in_utc = aware_datetime.astimezone(utc)

    naive_datetime_as_utc_converted_to_tz = tz.localize(naive_datetime)

except UnknownTimeZoneError:
    pass  # {handle error}
```

3.2 Getting a location's time zone offset

```
from datetime import datetime
from pytz import timezone, utc


def get_offset(*, lat, lng):
    """
    returns a location's time zone offset from UTC in minutes.
    """

    today = datetime.now()
    tz_target = timezone(tf.certain_timezone_at(lng=lng, lat=lat))
    # ATTENTION: tz_target could be None! handle error case
    today_target = tz_target.localize(today)
    today_utc = utc.localize(today)
    return (today_utc - today_target).total_seconds() / 60
```

(continues on next page)

(continued from previous page)

```
bergamo = {"lat": 45.69, "lng": 9.67}
minute_offset = get_offset(**bergamo)
```

also see the pytz Doc.

3.3 Django

querying the timezone name in a `Django` view:

```
def find_timezone(request, lat, lng):
    lat = float(lat)
    lng = float(lng)
    try:
        timezone_name = tf.timezone_at(lng=lng, lat=lat)
    except ValueError:
        # the coordinates were out of bounds
        pass  # {handle error}
    if timezone_name is None:
        # no timezone matched
        ...

    # do something with timezone_name
    ...
```

3.4 Use other data

3.4.1 File converter script

This package includes the `file_converter.py` script to parse timezone data and compile the binary data files required by the `timezonefinder` package. This script is built for processing the specific `geojson` format of the default data: timezone-boundary-builder. Any other data in this format can also be parsed:

```
python /path/to/timezonefinder/scripts/file_converter.py [-inp /path/to/input.json] [-
↪out /path/to/output_folder]
```

Per default the script parses the `combined.json` from its own parent directory (`timezonefinder`) into data files inside its parent directory. How to use the `timezonefinder` package with data files from another location is described *HERE*.

3.4.2 Data parsing shell script

The included `parse_data.sh` shell script simplifies downloading the latest version of timezone-boundary-builder data and parsing in with `file_converter.py`. It supports downloading and parsing the `timezone-boundary-builder` version WITHOUT ocean timezones. This is useful if you do not require ocean timezones and want to have smaller data files.

```
/bin/bash  /path/to/timezonefinder/parse_data.sh
```

CHAPTER

FOUR

PERFORMANCE

4.1 C extension

During installation `timezonefinder` automatically tries to compile a C extension with an implementation of the time critical point in polygon check algorithm. In order for this to work, a Clang compiler has to be installed.

Note: If compilation fails (due to e.g. missing C compiler or broken `cffi` installation) `timezonefinder` will silently fall back to a pure Python implementation (~400x slower, cf. *speed test results* below).

For testing if the compiled C implementation of the point in polygon algorithm is being used:

```
TimezoneFinder.using_clang_pip()  # returns True or False
```

4.2 Numba

Some of the utility function (cf. `utils.py`) can be JIT compiled automatically by installing the optional dependency `numba`:

```
pip install timezonefinder[numba]
```

It is highly recommended to install `Numba` for increased performance when the C extensions cannot be used (e.g. C compiler is not present at build time).

Note: If Numba can be imported, the JIT compiled Python version of the point in polygon check algorithm will be used instead of the C alternative as it is even faster (cf. *speed test results* below).

For testing if Numba is being used to JIT compile helper functions:

```
TimezoneFinder.using_numba()  # returns True or False
```

4.3 In memory mode

To speed up the computations at the cost of memory consumption and initialisation time, pass `in_memory=True` during initialisation. This causes all binary files to be read into memory.

```
tf = TimezoneFinder(in_memory=True)
```

4.4 Speed Benchmark Results

obtained on MacBook Pro (15-inch, 2017), 2,8 GHz Intel Core i7 and timezonefinder version `6.1.0`

4.4.1 Timezone finding

`scripts/check_speed_timezone_finding.py`

Without Numba (using C extension):

```
using C implementation: True
using Numba: False

10000 'on land points' (points included in a land timezone)
in memory mode: False

function name                          | s/query    | pts/s
-------------------------------------------------------
TimezoneFinder.timezone_at()           | 7.5e-05    | 1.3e+04
TimezoneFinder.timezone_at_land()      | 7.7e-05    | 1.3e+04
TimezoneFinderL.timezone_at()          | 7.3e-06    | 1.4e+05
TimezoneFinderL.timezone_at_land()     | 8.3e-06    | 1.2e+05

10000 random points (anywhere on earth)
in memory mode: False

function name                          | s/query    | pts/s
-------------------------------------------------------
TimezoneFinder.timezone_at()           | 8.8e-05    | 1.1e+04
TimezoneFinder.timezone_at_land()      | 8.9e-05    | 1.1e+04
TimezoneFinderL.timezone_at()          | 6.6e-06    | 1.5e+05
TimezoneFinderL.timezone_at_land()     | 9.5e-06    | 1.1e+05

10000 'on land points' (points included in a land timezone)
in memory mode: True

function name                          | s/query    | pts/s
-------------------------------------------------------
TimezoneFinder.timezone_at()           | 3.9e-05    | 2.6e+04
TimezoneFinder.timezone_at_land()      | 4.0e-05    | 2.5e+04
TimezoneFinderL.timezone_at()          | 6.3e-06    | 1.6e+05
TimezoneFinderL.timezone_at_land()     | 8.6e-06    | 1.2e+05
```

(continues on next page)

(continued from previous page)

```
10000 random points (anywhere on earth)
in memory mode: True

function name                          | s/query    | pts/s
-----------------------------------------------------------
TimezoneFinder.timezone_at()           | 3.5e-05    | 2.8e+04
TimezoneFinder.timezone_at_land()      | 3.9e-05    | 2.6e+04
TimezoneFinderL.timezone_at()          | 6.9e-06    | 1.5e+05
TimezoneFinderL.timezone_at_land()     | 9.0e-06    | 1.1e+05
```

With Numba:

```
using C implementation: False
using Numba: True

10000 'on land points' (points included in a land timezone)
in memory mode: False

function name                          | s/query    | pts/s
-----------------------------------------------------------
TimezoneFinder.timezone_at()           | 7.1e-05    | 1.4e+04
TimezoneFinder.timezone_at_land()      | 7.4e-05    | 1.3e+04
TimezoneFinderL.timezone_at()          | 6.5e-06    | 1.5e+05
TimezoneFinderL.timezone_at_land()     | 9.1e-06    | 1.1e+05

10000 random points (anywhere on earth)
in memory mode: False

function name                          | s/query    | pts/s
-----------------------------------------------------------
TimezoneFinder.timezone_at()           | 8.2e-05    | 1.2e+04
TimezoneFinder.timezone_at_land()      | 8.1e-05    | 1.2e+04
TimezoneFinderL.timezone_at()          | 6.9e-06    | 1.5e+05
TimezoneFinderL.timezone_at_land()     | 8.8e-06    | 1.1e+05

10000 'on land points' (points included in a land timezone)
in memory mode: True

function name                          | s/query    | pts/s
-----------------------------------------------------------
TimezoneFinder.timezone_at()           | 3.7e-05    | 2.7e+04
TimezoneFinder.timezone_at_land()      | 4.0e-05    | 2.5e+04
TimezoneFinderL.timezone_at()          | 6.9e-06    | 1.5e+05
TimezoneFinderL.timezone_at_land()     | 8.1e-06    | 1.2e+05

10000 random points (anywhere on earth)
in memory mode: True

function name                          | s/query    | pts/s
-----------------------------------------------------------
TimezoneFinder.timezone_at()           | 3.2e-05    | 3.1e+04
TimezoneFinder.timezone_at_land()      | 3.4e-05    | 2.9e+04
```

(continues on next page)

(continued from previous page)

```
TimezoneFinderL.timezone_at()       | 6.4e-06    | 1.6e+05
TimezoneFinderL.timezone_at_land()  | 7.6e-06    | 1.3e+05
```

4.4.2 Point in polygon checks

`scripts/check_speed_inside_polygon.py`

Without Numba:

```
testing the speed of the different point in polygon algorithm implementations
testing 1000 queries: random points and timezone polygons
Python implementation using Numba JIT compilation: False

inside_clang: 2.7e-05 s/query, 3.7e+04 queries/s
inside_python: 1.0e-02 s/query, 9.9e+01 queries/s
C implementation is 374.1x faster than the Python implementation WITHOUT Numba
```

With Numba:

```
testing the speed of the different point in polygon algorithm implementations
testing 10000 queries: random points and timezone polygons
Python implementation using Numba JIT compilation: True

inside_clang: 2.2e-05 s/query, 4.5e+04 queries/s
inside_python: 1.8e-05 s/query, 5.5e+04 queries/s
Python implementation WITH Numba is 0.2x faster than the C implementation
```

4.4.3 Initialisation

```
testing initialiation: TimezoneFinder(in_memory=True)
avg. startup time: 7.01e-01 (10 runs)

testing initialiation: TimezoneFinder(in_memory=False)
avg. startup time: 7.85e-01 (10 runs)

testing initialiation: TimezoneFinderL(in_memory=True)
avg. startup time: 6.66e-01 (10 runs)

testing initialiation: TimezoneFinderL(in_memory=False)
avg. startup time: 7.30e-01 (10 runs)
```

CHAPTER

FIVE

ABOUT

`timezonefinder` is a python package for looking up the corresponding timezone for given coordinates on earth entirely offline.

Timezones internally are being represented by polygons and the timezone membership of a given point (= lat lng coordinate pair) is determined by a point in polygon (PIP) check. In many cases an expensive PIP check can be avoided. This package currently uses a precomputed timezone polygon index based on the geospatial hexagon index of the h3 library. Among other tweaks this index makes `timezonefinder` efficient (also check the *performance chapter*). See the docstrings in the source code for further explanation.

5.1 Data

Current **data set** in use: precompiled timezone-boundary-builder (WITH oceans, geoJSON)

Note: In the data set the timezone polygons often include territorial waters -> they do NOT follow the shorelines. This makes the results of `certain_timezone_at()` less expressive: from a timezone match one cannot distinguish whether a query point lies on land or in ocean.

Note: Please note that timezone polygons might be overlapping (cf. e.g. timezone-boundary-builder/issue/105) and that hence a query coordinate can actually match multiple time zones. `timezonefinder` does currently NOT support such multiplicity and will always only return the first found match.

- package size: ~46 MB
- original data size: ~110 MB

5.2 References

GitHub

PyPI

online GUI and API

conda-forge feedstock

ruby port: timezone_finder

download stats

5.3 License

`timezonefinder` is distributed under the terms of the MIT license (see LICENSE).

5.4 Alternative python packages

- tzfpy (less accurate, more lightweight, faster)
- pytzwhere (not maintained)

5.5 Comparison to tzfpy

Differences:

- `tzfpy` is a Python binding of the Rust package `tzf-rs`
- `tzfpy` has no startup time
- **`tzfpy` uses simplified timezone polygons (data):**
  - this reduces the memory requirements
  - this reduces the accuracy
  - this increases the lookup speed
- `tzfpy` uses hierarchical tree of rectangles to speed up the lookup but auto fall back to polygon data if cache miss

5.6 Comparison to pytzwhere

This project has originally been derived from pytzwhere (github), but aims at providing improved performance and usability.

`pytzwhere` is parsing a 76MB .csv file (floats stored as strings!) completely into memory and computing shortcuts from this data on every startup. This is time, memory and CPU consuming. Additionally calculating with floats is slow, keeping those 4M+ floats in the RAM all the time is unnecessary and the precision of floats is not even needed in this case (s. detailed comparison and speed tests below).

In comparison most notably initialisation time and memory usage are significantly reduced. `pytzwhere` is using up to 450MB of RAM (with `shapely` and `numpy` active), because it is parsing and keeping all the timezone polygons in the

memory. This uses unnecessary time/ computation/ memory and this was the reason I created this package in the first place. This package uses at most 40MB (= encountered memory consumption of the python process) and has some more advantages:

Differences:

- highly decreased memory usage
- highly reduced start up time
- usage of 32bit int (instead of 64+bit float) reduces computing time and memory consumption. The accuracy of 32bit int is still high enough. According to my calculations the worst accuracy is 1cm at the equator. This is far more precise than the discrete polygons in the data.
- the data is stored in memory friendly binary files (approx. 41MB in total, original data 120MB .json)
- data is only being read on demand (not completely read into memory if not needed)
- precomputed shortcuts are included to quickly look up which polygons have to be checked
- function `get_geometry()` enables querying timezones for their geometric shape (= multipolygon with holes)
- further speedup possible by the use of `numba` (code JIT compilation)
- tz_where is still using the outdated tz_world data set

5.7 Contact

Tell me if and how your are using this package. This encourages me to develop and test it further.

Most certainly there is stuff I missed, things I could have optimized even further or explained more clearly, etc. I would be really glad to get some feedback.

If you encounter any bugs, have suggestions etc. do not hesitate to **open an Issue** or **add a Pull Requests** on Git. Please refer to the *contribution guidelines*

5.8 Acknowledgements

Thanks to:

- Adam for adding organisational features to the project and for helping me with publishing and testing routines.
- ringsaturn for valuable feedback, sponsoring this project, creating the `tzfpy` package and adding the `pytz` compatibility extra
- snowman2 for creating the conda-forge recipe.
- synapticarbors for fixing Numba import with py27.
- zedrdave for valuable feedback.
- Tyler Huntley for adding docstrings

CHAPTER

SIX

API DOCUMENTATION

6.1 TimezoneFinderL

class timezonefinder.**TimezoneFinderL**(*bin_file_location: str | Path | None = None*, *in_memory: bool = False*)

Bases: AbstractTimezoneFinder

a 'light' version of the TimezoneFinder class for quickly suggesting a timezone for a point on earth

Instead of using timezone polygon data like TimezoneFinder, this class only uses a precomputed 'shortcut' to suggest a probable result: the most common zone in a rectangle of a half degree of latitude and one degree of longitude

timezone_at(*, *lng: float*, *lat: float*) → str | None

instantly returns the name of the most common zone within the corresponding shortcut

Note: 'most common' in this context means that the polygons with the most coordinates in sum occurring in the corresponding shortcut belong to this zone.

Parameters

- **lng** – longitude of the point in degree (-180.0 to 180.0)
- **lat** – latitude in degree (90.0 to -90.0)

Returns

the timezone name of the most common zone or None if there are no timezone polygons in this shortcut

bin_file_location: Path

shortcut_mapping

in_memory

timezone_names

poly_zone_ids

__init__(*bin_file_location: str | Path | None = None*, *in_memory: bool = False*)

Initialize the AbstractTimezoneFinder. :param bin_file_location: The path to the binary data files to use. If None, uses native package data. :param in_memory: Whether to completely read and keep the binary files in memory.

`binary_data_attributes: List[str] = ['poly_zone_ids']`

List of attribute names that store opened binary data files.

`get_shortcut_polys`(*, *lng: float*, *lat: float*) → ndarray

Get the polygon IDs in the shortcut corresponding to the given coordinates.

Parameters

- **lng** – The longitude of the point in degrees (-180.0 to 180.0).
- **lat** – The latitude of the point in degrees (90.0 to -90.0).

Returns
An array of polygon IDs.

`most_common_zone_id`(*, *lng: float*, *lat: float*) → int | None

Get the most common zone ID in the shortcut corresponding to the given coordinates.

Parameters

- **lng** – The longitude of the point in degrees (-180.0 to 180.0).
- **lat** – The latitude of the point in degrees (90.0 to -90.0).

Returns
The most common zone ID or None if no polygons exist in the shortcut.

`property nr_of_zones`

Get the number of timezones.

Return type
int

`timezone_at_land`(*, *lng: float*, *lat: float*) → str | None

computes in which land timezone a point is included in

Especially for large polygons it is expensive to check if a point is really included. To speed things up there are "shortcuts" being used (stored in a binary file), which have been precomputed and store which timezone polygons have to be checked.

Parameters

- **lng** – longitude of the point in degree (-180.0 to 180.0)
- **lat** – latitude in degree (90.0 to -90.0)

Returns
the timezone name of a matching polygon or `None` when an ocean timezone ("Etc/GMT+-XX") has been matched.

`unique_timezone_at`(*, *lng: float*, *lat: float*) → str | None

returns the name of a unique zone within the corresponding shortcut

Parameters

- **lng** – longitude of the point in degree (-180.0 to 180.0)
- **lat** – latitude in degree (90.0 to -90.0)

Returns
the timezone name of the unique zone or `None` if there are no or multiple zones in this shortcut

unique_zone_id(*, *lng: float*, *lat: float*) → int | None

Get the unique zone ID in the shortcut corresponding to the given coordinates.

Parameters

- **lng** – The longitude of the point in degrees (-180.0 to 180.0).
- **lat** – The latitude of the point in degrees (90.0 to -90.0).

Returns
The unique zone ID or None if no polygons exist in the shortcut.

static using_clang_pip() → bool

Returns
True if the compiled C implementation of the point in polygon algorithm is being used

static using_numba() → bool

Check if Numba is being used.

Return type
bool

Returns
True if Numba is being used to JIT compile helper functions

zone_id_of(*poly_id: int*) → int

Get the zone ID of a polygon.

Parameters
poly_id (*int*) – The ID of the polygon.

Return type
int

zone_ids_of(*poly_ids: ndarray*) → ndarray

Get the zone IDs of multiple polygons.

Parameters
poly_ids – An array of polygon IDs.

Returns
An array of zone IDs corresponding to the given polygon IDs.

zone_name_from_id(*zone_id: int*) → str

Get the zone name from a zone ID.

Parameters
zone_id – The ID of the zone.

Returns
The name of the zone.

Raises
ValueError – If the timezone could not be found.

zone_name_from_poly_id(*poly_id: int*) → str

Get the zone name from a polygon ID.

Parameters
poly_id – The ID of the polygon.

Returns
The name of the zone.

6.2 TimezoneFinder

class timezonefinder.**TimezoneFinder**(*bin_file_location: str | None = None*, *in_memory: bool = False*)

Bases: AbstractTimezoneFinder

Class for quickly finding the timezone of a point on earth offline.

Because of indexing ("shortcuts"), not all timezone polygons have to be tested during a query.

Opens the required timezone polygon data in binary files to enable fast access. For a detailed documentation of data management please refer to the code documentation of file_converter.py

Variables
binary_data_attributes – the names of all attributes which store the opened binary data files

Parameters

- **bin_file_location** – path to the binary data files to use, None if native package data should be used
- **in_memory** – whether to completely read and keep the binary files in memory

binary_data_attributes: List[str] = ['poly_zone_ids', 'poly_coord_amount', 'poly_adr2data', 'poly_bounds', 'poly_data', 'poly_nr2zone_id', 'hole_coord_amount', 'hole_adr2data', 'hole_data']

List of attribute names that store opened binary data files.

__init__(*bin_file_location: str | None = None*, *in_memory: bool = False*)

Initialize the AbstractTimezoneFinder. :param bin_file_location: The path to the binary data files to use. If None, uses native package data. :param in_memory: Whether to completely read and keep the binary files in memory.

property nr_of_polygons: int

Get the number of polygons.

Returns
The number of polygons.

coords_of(*polygon_nr: int = 0*) → ndarray

Get the coordinates of a polygon.

Parameters
polygon_nr – The index of the polygon.

Returns
Array of coordinates.

get_polygon(*polygon_nr: int*, *coords_as_pairs: bool = False*) → List[List[Tuple[float, float]] | List[List[float]]]

Get the polygon coordinates of a given polygon number.

Parameters

- **polygon_nr** – Polygon number
- **coords_as_pairs** – Determines the structure of the polygon representation

Returns
List of polygon coordinates

get_geometry(*tz_name: str | None = ''*, *tz_id: int | None = 0*, *use_id: bool = False*, *coords_as_pairs: bool = False*)

retrieves the geometry of a timezone polygon

Parameters

- **tz_name** – one of the names in `timezone_names.json` or `self.timezone_names`
- **tz_id** – the id of the timezone (=index in `self.timezone_names`)
- **use_id** – if `True` uses `tz_id` instead of `tz_name`
- **coords_as_pairs** – determines the structure of the polygon representation

Returns

a data structure representing the multipolygon of this timezone output format: `[ [polygon1, hole1, hole2...], [polygon2, ...], ...]` and each polygon and hole is itself formatted like: `([longitudes], [latitudes])` or `[(lng1,lat1), (lng2,lat2),...]` if `coords_as_pairs=True`.

get_polygon_boundaries(*poly_id: int*) → Tuple[int, int, int, int]

returns the boundaries of the polygon = (lng_max, lng_min, lat_max, lat_min) converted to int32

outside_the_boundaries_of(*poly_id: int*, *x: int*, *y: int*) → bool

Check if a point is outside the boundaries of a polygon.

Parameters

- **poly_id** – Polygon ID
- **x** – X-coordinate of the point
- **y** – Y-coordinate of the point

Returns

True if the point is outside the boundaries, False otherwise

inside_of_polygon(*poly_id: int*, *x: int*, *y: int*) → bool

Check if a point is inside a polygon.

Parameters

- **poly_id** – Polygon ID
- **x** – X-coordinate of the point
- **y** – Y-coordinate of the point

Returns

True if the point is inside the polygon, False otherwise

poly_zone_ids

poly_coord_amount

poly_adr2data

poly_bounds

poly_data

poly_nr2zone_id

hole_coord_amount

hole_adr2data

hole_data

hole_registry

bin_file_location: Path

get_shortcut_polys(*, *lng: float*, *lat: float*) → ndarray

Get the polygon IDs in the shortcut corresponding to the given coordinates.

Parameters

- **lng** – The longitude of the point in degrees (-180.0 to 180.0).
- **lat** – The latitude of the point in degrees (90.0 to -90.0).

Returns
An array of polygon IDs.

in_memory

most_common_zone_id(*, *lng: float*, *lat: float*) → int | None

Get the most common zone ID in the shortcut corresponding to the given coordinates.

Parameters

- **lng** – The longitude of the point in degrees (-180.0 to 180.0).
- **lat** – The latitude of the point in degrees (90.0 to -90.0).

Returns
The most common zone ID or None if no polygons exist in the shortcut.

property nr_of_zones

Get the number of timezones.

Return type
int

shortcut_mapping

timezone_at(*, *lng: float*, *lat: float*) → str | None

computes in which ocean OR land timezone a point is included in

Especially for large polygons it is expensive to check if a point is really included. In case there is only one possible zone (left), this zone will instantly be returned without actually checking if the query point is included in this polygon.

To speed things up there are "shortcuts" being used
which have been precomputed and store which timezone polygons have to be checked.

Note: Since ocean timezones span the whole globe, some timezone will always be matched! *None* can only be returned when you have compiled timezone data without such "full coverage".

Parameters

- **lng** – longitude of the point in degree (-180.0 to 180.0)
- **lat** – latitude in degree (90.0 to -90.0)

Returns
the timezone name of the matched timezone polygon. possibly "Etc/GMT+-XX" in case of an ocean timezone.

`timezone_at_land`(*, *lng: float*, *lat: float*) → str | None

computes in which land timezone a point is included in

Especially for large polygons it is expensive to check if a point is really included. To speed things up there are "shortcuts" being used (stored in a binary file), which have been precomputed and store which timezone polygons have to be checked.

Parameters

- `lng` – longitude of the point in degree (-180.0 to 180.0)
- `lat` – latitude in degree (90.0 to -90.0)

Returns
the timezone name of a matching polygon or `None` when an ocean timezone ("Etc/GMT+-XX") has been matched.

`timezone_names`

`unique_timezone_at`(*, *lng: float*, *lat: float*) → str | None

returns the name of a unique zone within the corresponding shortcut

Parameters

- `lng` – longitude of the point in degree (-180.0 to 180.0)
- `lat` – latitude in degree (90.0 to -90.0)

Returns
the timezone name of the unique zone or `None` if there are no or multiple zones in this shortcut

`unique_zone_id`(*, *lng: float*, *lat: float*) → int | None

Get the unique zone ID in the shortcut corresponding to the given coordinates.

Parameters

- `lng` – The longitude of the point in degrees (-180.0 to 180.0).
- `lat` – The latitude of the point in degrees (90.0 to -90.0).

Returns
The unique zone ID or None if no polygons exist in the shortcut.

`static using_clang_pip`() → bool

Returns
True if the compiled C implementation of the point in polygon algorithm is being used

`static using_numba`() → bool

Check if Numba is being used.

Return type
bool

Returns
True if Numba is being used to JIT compile helper functions

zone_id_of(*poly_id: int*) → int

Get the zone ID of a polygon.

Parameters
poly_id (*int*) – The ID of the polygon.

Return type
int

zone_ids_of(*poly_ids: ndarray*) → ndarray

Get the zone IDs of multiple polygons.

Parameters
poly_ids – An array of polygon IDs.

Returns
An array of zone IDs corresponding to the given polygon IDs.

zone_name_from_id(*zone_id: int*) → str

Get the zone name from a zone ID.

Parameters
zone_id – The ID of the zone.

Returns
The name of the zone.

Raises
ValueError – If the timezone could not be found.

zone_name_from_poly_id(*poly_id: int*) → str

Get the zone name from a polygon ID.

Parameters
poly_id – The ID of the polygon.

Returns
The name of the zone.

certain_timezone_at(*, *lng: float*, *lat: float*) → str | None

checks in which timezone polygon the point is certainly included in

---

Note: this is only meaningful when you have compiled your own timezone data where there are areas without timezone polygon coverage. Otherwise, some timezone will always be matched and the functionality is equal to using *.timezone_at()* -> useless to actually test all polygons.

Note: using this function is less performant than *.timezone_at()*

Parameters

- **lng** – longitude of the point in degree
- **lat** – latitude in degree

Returns
the timezone name of the polygon the point is included in or *None*

CHAPTER

SEVEN

CONTRIBUTION GUIDELINES

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

7.1 Types of Contributions

7.1.1 Report Bugs

Report bugs via Github Issues.

If you are reporting a bug, please include:

- Your version of this package, python and Numba (if you use it)
- Any other details about your local setup that might be helpful in troubleshooting, e.g. operating system.
- Detailed steps to reproduce the bug.
- Detailed description of the bug (error log etc.).

7.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

7.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "help wanted" and not assigned to anyone is open to whoever wants to implement it - please leave a comment to say you have started working on it, and open a pull request as soon as you have something working, so that Travis starts building it.

Issues without "help wanted" generally already have some code ready in the background (maybe it's not yet open source), but you can still contribute to them by saying how you'd find the fix useful, linking to known prior art, or other such help.

7.1.4 Write Documentation

Probably for some features the documentation is missing or unclear. You can help with that!

7.1.5 Submit Feedback

The best way to send feedback is to file an issue via Github Issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement. Create multiple issues if necessary.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

7.2 Get Started!

Ready to contribute? Here's how to set up this package for local development.

- Fork this repo on GitHub.
- Clone your fork locally
- To make changes, create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

- Check out the instructions and notes in `publish.py`
- Install `tox` and run the tests:

```
$ pip install tox
$ tox
```

  The `tox.ini` file defines a large number of test environments, for different Python etc., plus for checking codestyle. During development of a feature/fix, you'll probably want to run just one plus the relevant codestyle:

```
$ tox -e codestyle
```

- Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

- Submit a pull request through the GitHub website. This will trigger the Travis CI build which runs the tests against all supported versions of Python.

CHAPTER

EIGHT

CHANGELOG

8.1 6.5.0 (2024-03-14)

- updated the data to 2024a.

internal:

- use ruff linter in pre-commit hook
- make dependency specifications less strict

8.2 6.4.1 (2024-02-08)

- added official support for python 3.8 again, by specifying numba as multiple constraint dependency

internal:

- added unit tests for polygon boundary binary reading

8.3 6.4.0 (2024-02-02)

- added python 3.12 support (supported by numba since release 0.59.0), Closes #208
- dropped official support for python 3.8, because the optional dependency numba requires python 3.9. this package might still work with python 3.8, but it is not tested anymore.

8.4 6.3.0 (2024-02-01)

- updated the data to 2023d.

internal:

- added docstrings. Thanks to Tyler Huntley
- automatically skip GitHub actions publishing when the version already exists. useful for minor improvements without publishing a version. build would always fail otherwise
- enable tests for python 3.11 with numba
- enable tests for python 3.12
- added tests for generating the documentation

- use poetry dependency group specification (closing #199)

8.5 6.2.0 (2023-03-26)

- updated the data to 2023b.

8.6 6.1.10 (2023-03-22)

- added a *pytz* extra for easily maintaining compatibility
- improved documentation

8.7 6.1.9 (2022-12-06)

- updated the data to 2022g.

8.8 6.1.8 (2022-11-25)

- pumped `h3` dependency to `>=3.7.6,<4` to support python 3.11 (FIX #170)
- added python 3.11 tests (not yet supporting numba)

8.9 6.1.7 (2022-11-20)

- updated the data to 2022f.
- pinning dependencies more strictly

8.10 6.1.6 (2022-10-30)

- updated the data to 2022d.

8.11 6.1.5 (2022-10-25)

- updated the data to 2022b.
- logging build failures with warnings

8.12 6.1.4 (2022-10-23)

- more permissive optional `Numba` dependency specification (FIX #162, impossible using latest numpy version)
- made all dependency specifications more permissive following the same rationale

8.13 6.1.3 (2022-09-23)

- bugfix broken package build in the case of a broken `cffi` installation (GitHub issue #155). Skip build process if `cffi` fails. For performance reasons using the C extension should remain the default behavior. Hence the `cffi` dependency should not be optional.

8.14 6.1.2 (2022-09-13)

- bugfix potentially broken pip install due to a mismatch in `cffi` versions (GitHub issue #151)

8.15 6.1.1 (2022-08-18)

internals:

- **minimized and cleaned up installation footprint (addresses GitHub Issue #151):**
  - excluded script, changelog etc. files
  - included C extension into the "timezonefinder" package folder
- added initialisation speed benchmark

8.16 6.1.0 (2022-08-15)

- included point-in-polygon implementation in C
- included build script to (optionally) build C point-in-polygon extension automatically during installation
- added `cffi` as a dependency to build and interact with the C extension
- improved initialisation speed: read timezone polygon id index (h3 mapping) with `np.fromfile`
- improved CLI speed: construct TimezoneFinder() instances only on demand

internals:

- updated documentation: `Numba` installation is no longer recommended (it is a huge dependency and should be optional)
- clarified documentation: TimezoneFinder() instances should be reused
- added separate speed benchmark scripts for point in polygon algorithm implementations and the different timezone finding functions
- added separate section in the documentation for performance including speed benchmark results
- added checks if all timezone polygons are actually in use (appear in index) to the file conversion script
- added and improved utility functions as well as tests

- improved typing

8.17 6.0.2 (2022-07-08)

- bump numpy dependency version to `1.22` (vulnerability fix)
- officially supported python versions `>=3.8,<3.11` (due to numpy and numba constraints)
- packaging now completely based on pyproject.toml (poetry)

8.18 6.0.1 (2022-05-20)

- explicitly included `py.typed` in the package to allow mypy users to run static type checking

8.19 6.0.0 (2022-05-09)

breaking changes:

- new dependency: using h3 for indexing the timezone polygons to check ("shortcuts) instead of the previous own indexing implementation. technical details: storing all 41,162 hex cells at resolution 3 and the corresponding timezone polygons which appear in them in the `shortcuts.bin` (~500 KB).
- removed `.closest_timezone_at()`: with the current data set with ocean zones in use, any point is included in some zone. it is therefore not meaningful to search for the closest boundary! Also the timezone polygons do NOT follow the shorelines. This makes the results of `closest_timezone_at()` somewhat less expressive. Maintaining the non-trivial distance computation algorithms is not really at the core responsibility of this package.
- officially only supporting `python>=3.7` (removed official support for `python3.6`, since the `numpy` dependency did so)
- removed `v` from the github release/version tags

internals:

- updated the data to 2021c. please note that timezone polygons might be overlapping (cf. e.g. timezone-boundary-builder/issue/105) and that hence a query coordinate can actually match multiple time zones. `timezonefinder` does currently NOT support such multiplicity and will always only return the first found match.
- shortcuts: sorting according to size of polygons (amount of coordinates) instead of the count of zone ids. useful as optimisation: smaller polygons will be checked first and can hence be "ruled-out" faster
- "most common": now meaning the zone with the largest polygons in the shortcut (last in the shortcut sorting). please note that this does not necessarily mean the most area in the shortcut is covered by this zone. the polygon size is just an easier to compute heuristic.
- officially supporting python versions >=3.7,<3.11 (like `numba`)
- using poetry for dependency management
- using GitHub actions for CI instead of travis
- some minor typing improvements
- pre-commit hook improvements

In case you have criticism or feedback please reach out by creating an issue, discussion or PR on GitHub.

8.20 5.2.0 (2021-02-09)

- added function `unique_timezone_at()` (based on the request in issue #112). Allows querying for the unique zone within the corresponding shortcut.

8.21 5.1.1 (2021-02-03)

- BUGFIX: get_geometry() now also works for the last zone
- add get_geometry() tests
- black code style
- pre-commit checks

8.22 5.1.0 (2021-01-14)

- update the command line interface. the package can now directly be called with `timezonefinder`
- added the new query functions to the command line interface (to match the online API)

8.23 5.0.0 (2020-12-23)

MAJOR CHANGES:

Due to multiple user requests the ocean timezones ("Etc/GMT+-XX") are now included in the data files per default. fix #88. Since ocean timezones span the whole globe, now every point lies within a timezone!

API changes: * added `timezone_at_land()`: replaces the previous `timezone_at()` and returns `None` in case of a matched ocean timezone.

- deprecated `certain_timezone_at()`. only meaningful in the case of timezone data WITHOUT oceans. Has equal results as `timezone_at()`, but is more expensive to use.
- also looking a single closest timezone boundary with `closest_timezone_at()` is not really meaningful, since every point lies within a zone!
- refactored tests. new test cases for ocean timezones

8.24 4.5.0 (2020-11-06)

BUGFIX: handle output destination for data files correctly in file_converter.py (FIX #107)

- updated the data to 2020d
- disable a test case for an Uzbek enclave. tests fail at this coordinate, possibly a bug. issue filed here: https://github.com/evansiroky/timezone-boundary-builder/issues/94
- update parse_data.sh script to properly handle new data format

8.25 4.4.1 (2020-08-04)

BUGFIX: a longitude of 180 equals -180 (not 0.0 as previously implemented)

8.26 4.4.0 (2020-05-14)

- added new class TimezonefinderL for using JUST shortcuts (without timezone polygon data)
- therefore included the most common timezone of each shortcut stored in the binary file `shortcuts_direct_id.bin`
- introduced typing
- included API documentation
- read hole registry directly from json, `hole_poly_ids.bin` not required any more
- added the `parse_data.sh` shell script for downloading the latest timezone data, also with oceans

improvements of file_converter.py:

- added command line arguments for specifying the input and output directories
- read binary names from `global_settings.py`
- read data types from `global_settings.py`
- use with statement for writing binaries
- automatically detect overflow for each data type in use
- cleanup code, remove redundancies, improve codestyle
- fixing #101: make imports work for local and remote execution

8.27 4.3.1 (2020-04-29)

- BUGFIX #99: include the correct timezone_names.json in build
- wheel specific for the supported python versions (3.6, 3.7, 3.8)

8.28 4.3.0 (2020-04-28)

- updated the data to 2020a
- added "extra" simplifying the installation of Numba
- added minimal required python version
- added minimal required version of the dependencies
- simplified and updated settings (e.g. reading current version from file)
- also testing python 3.8 now
- loading version from file

8.29 4.2.0 (2019-12-15)

- added option to specify the location of the binary data files to use. making it possible to easily point to own compiled data. also load timezone names json from this location
- make timezone names a class attribute (instead of a global variable)
- simplify code for opening and closing multiple binary files
- added tests for a specified path to the data
- testing multiple python3 versions automatically
- pinned new requirements
- importlib_resources removed from the dependencies
- added a documentation at: https://timezonefinder.readthedocs.io/en/latest/
- added contribution guidelines

8.30 4.1.0 (2019-07-07)

- updated the data to 2019b
- added description of using vectorized input in readme

8.31 4.0.3 (2019-06-23)

- clarification of readme: referenced latest *timezonefinderL* release, better rst headlines, updated shield.io banner syntax
- clarification of speedup times (exponential notation)
- removed *six* and py2 dependency from tests
- minor updates to publishing routine
- minor improvement in timezone_at(): conversion coordinates to int later only when required

8.32 4.0.2 (2019-04-01)

- updated the data to 2019a

8.33 4.0.1 (2019-03-12)

- BUGFIX: fixing #77 (missing dependency in setup.py)

8.34 4.0.0 (2019-03-12)

- ATTENTION: Dropped Python2 support (#72)! *six* dependency no longer required.
- BUGFIX: fixing #74 (broken py3 with numba support)
- added *in_memory*-mode (adapted unit tests to test both modes, added speed tests and explanation to readme)
- use of timeit in speed tests for more accurate results
- dropped use of kwargs_only decorator (can be implemented directly with python3)

8.35 3.4.2 (2019-01-15)

- BUGFIX: fixing #70 (broken py2.7 with numba support)
- added automatic tox tests for py2.7 py3 environments with numba installed
- fixed coverage report

8.36 3.4.1 (2019-01-13)

- added test cases for the Numba helpers (#55)
- added more polygon tests to test the function inside_polygon()
- added global data type definitions (format strings) to `global_settings.py`
- removed tzwhere completely from the main tests (no comparison any more).
- removed code drafts for ahead of time compilation (#40)

8.37 3.4.0 (2019-01-06)

- updated the data to 2018i
- introduced `global_settings.py` to globally define settings and get rid of "magic numbers".

8.38 3.3.0 (2018-11-17)

- updated the data to 2018g

8.39 3.2.1 (2018-10-30)

- ATTENTION: the package `importlib_resources` is now required
- fixing automatic Conda build by exchanging `pkg_resources.resource_stream` with `importlib_resources.open_binary`
- added tests for overflow in helpers.py/inside_polygon()

8.40 3.2.0 (2018-10-23)

- ATTENTION: the package kwargs_only is not a requirement any more!
- fixing #63 (kwargs_only not in conda) enabling automatic conda forge builds by directly providing the kwargs_only functionality again
- added example.py with the code examples from the readme
- fixing #62 (overflow happening because of using numpy.int32): forcing int64 type conversion

8.41 3.1.0 (2018-09-27)

- fixing typo in requirements.txt
- updated publishing routine: reminder to include all direct dependencies and to compile the requirements.txt with python 2 (pip-tools)

8.42 3.0.2 (2018-09-26)

- ATTENTION: the package kwargs_only is now required! This functionality has previously been implemented by the author directly within this package, but some code features got deprecated.
- updated build/testing/publishing routine
- fixing issue #61 (six dependency not listed in setup.py)
- no more default arguments for timezone_at() and certain_timezone_at()
- no more comparison to (py-)tzwhere in the tests (test_it.py)
- updated requirements.txt (removed tzwhere and dependencies)
- prepared helpers_test.py for also testing helpers_numba.py
- exchanged deprecated inspect.getargspec() into .getfullargspec() in functional.py

8.43 3.0.1 (2018-05-30)

- fixing minor issue #58 (readme not rendering in pyPI)

8.44 3.0.0 (2018-05-17)

- ATTENTION: the package six is now required! (was necessary because of the new testing routine. improves compatibility standards)
- updated build/testing/publishing routine
- updated the data to 2018d
- fixing minor issue #52 (shortcuts being out of bounds for extreme coordinate values)
- **the list of polygon ids in each shortcut is sorted after freq. of appearance of their zone id.**
  this is critical for ruling out zones faster (as soon as just polygons of one zone are left this zone can be returned)
- using argparse package now for parsing the command line arguments
- added option of choosing between functions timezone_at() and certain_timezone_at() on the command line with flag -f
- the timezone names are now being stored in a readable JSON file
- adjusted the main test cases
- corrections and clarifications in the readme and code comments

8.45 2.1.2 (2017-11-20)

- bugfix: possibly uninitialized variable in closest_timezone_at()

8.46 2.1.1 (2017-11-20)

- updated the data to 2017c
- minor improvements in code style and readme
- include publishing routine script

8.47 2.1.0 (2017-05-19)

- updated the data to 2017a (tz_world is not being maintained any more)
- the file_converter has been updated to parse the new format of .json files
- the new data is much bigger (based on OSM Data, +40MB). I am sorry for this but its still better than small outdated data!
- in case size and speed matter more you than actuality, you can still check out older versions of timezonefinder(L)

- the new timezone polygons are not limited to the coastlines, but they are including some large parts of the sea. This makes the results of closest_timezone_at() somewhat meaningless (as with timezonefinderL).
- the polygons can not be simplified much more and as a consequence timezonefinderL is not being updated any more.
- simplification functions (used for compiling the data for timezonefinderL) have been deleted from the file_converter
- the readme has been updated to inform about this major change
- some tests have been temporarily disabled (with tzwhere still using a very old version of tz_world, a comparison does not make too much sense atm)

8.48 2.0.1 (2017-04-08)

- added missing package data entries (2.0.0 didn't include all necessary .bin files)

8.49 2.0.0 (2017-04-07)

- **ATTENTION: major change!: there is a second version of timezonefinder now: timezonefinderL. There the data has been simplified**
  for increasing speed reducing data size. Around 56% of the coordinates of the timezone polygons have been deleted there. Around 60% of the polygons (mostly small islands) have been included in the simplified polygons. For any coordinate on landmass the results should stay the same, but accuracy at the shorelines is lost. This eradicates the usefulness of closest_timezone_at() and certain_timezone_at() but the main use case for this package (= determining the timezone of a point on landmass) is improved. In this repo timezonefinder will still be maintained with the detailed (unsimplified) data.
- file_converter.py has been complemented and modified to perform those simplifications
- introduction of new function get_geometry() for querying timezones for their geometric shape
- added shortcuts_unique_id.bin for instantly returning an id if the shortcut corresponding to the coords only contains polygons of one zone
- data is now stored in separate binaries for ease of debugging and readability
- polygons are stored sorted after their timezone id and size
- timezonefinder can now be called directly as a script (experimental with reduced functionality, cf. readme)
- optimisations on point in polygon algorithm
- small simplifications in the helper functions
- clarification of the readme
- clarification of the comments in the code
- referenced the new conda-feedstock in the readme
- referenced the new timezonefinder API/GUI

8.50 1.5.7 (2016-07-21)

- ATTENTION: API BREAK: all functions are now keyword-args only (to prevent lng lat mix-up errors)
- fixed a little bug with too many arguments in a @jit function
- clarified usage of the package in the readme
- prepared the usage of the ahead of time compilation functionality of Numba. It is not enabled yet.
- sorting the order of polygons to check in the order of how often their zones appear, gives a speed bonus (for closest_timezone_at)

8.51 1.5.6 (2016-06-16)

- using little endian encoding now
- introduced test for checking the proper functionality of the helper functions
- wrote tests for proximity algorithms
- improved proximity algorithms: introduced exact_computation, return_distances and force_evaluation functionality (s. Readme or documentation for more info)

8.52 1.5.5 (2016-06-03)

- using the newest version (2016d, May 2016) of the tz world data
- holes in the polygons which are stored in the tz_world data are now correctly stored and handled
- rewrote the file_converter for storing the holes at the end of the timezone_data.bin
- added specific test cases for hole handling
- made some optimizations in the algorithms

8.53 1.5.4 (2016-04-26)

- using the newest version (2016b) of the tz world data
- rewrote the file_converter for parsing a .json created from the tz_worlds .shp
- had to temporarily fix one polygon manually which had the invalid TZID: 'America/Monterey' (should be 'America/Monterrey')
- had to make tests less strict because tzwhere still used the old data at the time and some results were simply different now

8.54 1.5.3 (2016-04-23)

- using 32-bit ints for storing the polygons now (instead of 64-bit): I calculated that the minimum accuracy (at the equator) is 1cm with the encoding being used. Tests passed.
- Benefits: 18MB file instead of 35MB, another 10-30% speed boost (depending on your hardware)

8.55 1.5.2 (2016-04-20)

- added python 2.7.6 support: replaced strings in unpack (unsupported by python 2.7.6 or earlier) with byte strings
- timezone names are now loaded from a separate file for better modularity

8.56 1.5.1 (2016-04-18)

- **added python 2.7.8+ support:**
  Therefore I had to change the tests a little bit (some operations were not supported). This only affects output. I also had to replace one part of the algorithms to prevent overflow in Python 2.7

8.57 1.5.0 (2016-04-12)

- automatically using optimized algorithms now (when numba is installed)
- added TimezoneFinder.using_numba() function to check if the import worked

8.58 1.4.0 (2016-04-07)

- **Added the `file_converter.py` to the repository: It converts the .csv from pytzwhere to another `.csv` and this one into the used `.bin`.**
  Especially the shortcut computation and the boundary storage in there save a lot of reading and computation time, when deciding which timezone the coordinates are in. It will help to keep the package up to date, even when the timezone data should change in the future.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

Symbols

B

C

G

H

I

M

N

O

P

S

T

U

Z