timezonefinder

Release 4.4.0

Jun 15, 2020

Contents:

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

CHAPTER 1

Getting started

1.1 Installation

Installation with conda: see instructions at conda-forge feedstock

Installation with pip: in your command line:

```
pip install timezonefinder
```

If the vanilla Python code is too slow for you, the time critical algorithms (in `helpers_numba.py`) can be automatically JIT compiled by `numba`. This speeds things up by a factor of around 100 (cf. *speed test results*).

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

1.2 Dependencies

`python3.6+`, `numpy`, (`numba`)

1.3 Basics

```
from timezonefinder import TimezoneFinder

tf = TimezoneFinder()
latitude, longitude = 52.5061, 13.358
tf.timezone_at(lng=longitude, lat=latitude) # returns 'Europe/Berlin'
```

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

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

CHAPTER 2

Usage

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

2.1 Initialisation

Create a new instance of the *TimezoneFinder class* to allow fast consequent timezone queries:

```
from timezonefinder import TimezoneFinder

tf = TimezoneFinder()
```

To save computing time at the cost of memory consumption and initialisation time pass `in_memory=True`. This causes all binary files to be read into memory. See the *speed test results*.

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

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')
```

For testing if the import of the JIT compiled algorithms worked:

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

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.

```
latitude, longitude = 52.5061, 13.358
tf.timezone_at(lng=longitude, lat=latitude) # returns 'Europe/Berlin'
```

Note:

- to avoid mixing up the arguments latitude and longitude have to be given as keyword arguments
- this function is optimized for speed and the common case to only query points within a timezone. The last possible timezone in proximity is always returned (without checking if the point is really included). So results might be misleading for points outside of any timezone.

For even faster results use *TimezoneFinderL*.

2.3 certain_timezone_at()

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.

```
tf.certain_timezone_at(lng=longitude, lat=latitude) # returns 'Europe/Berlin'
```

Note: The timezone polygons do NOT follow the shoreline. Consequently even if certain_timezone_at() does not return `None`, a query point could be at sea.

2.4 closest_timezone_at()

This function computes and compares the distances to the timezone polygon boundaries (expensive!). By default the function returns the closest timezone of all polygons within +-1 degree lng and +-1 degree lat (or None).

```
longitude = 12.773955
latitude = 55.578595
tf.closest_timezone_at(lng=longitude, lat=latitude) # returns 'Europe/Copenhagen'
```

Note:

- This function does not check whether a point is included in a timezone polygon.
- The timezone polygons do NOT follow the shoreline. This causes the computed distance from a timezone polygon to be not really accurate!

Options:

To increase search radius even more, use the `delta_degree`-option:

```
tf.closest_timezone_at(lng=longitude, lat=latitude, delta_degree=3)
```

This checks all the polygons within +-3 degree lng and +-3 degree lat. I recommend only slowly increasing the search radius, since computation time increases quite quickly (with the amount of polygons which need to be evaluated). When you want to use this feature a lot, consider using `Numba` to save computing time.

Note: x degrees lat are not the same distance apart than x degree lng (earth is a sphere)! As a consequence getting a result does NOT mean that there is no closer timezone! It might just not be within the area (given in degree!) being queried.

With `exact_computation=True` the distance to every polygon edge is computed (way more complicated), instead of just evaluating the distances to all the vertices. This only makes a real difference when the boundary of a polygon is very close to the query point.

With `return_distances=True` the output looks like this:

```
( 'tz_name_of_the_closest_polygon',[ distances to every polygon in km], [tz_names of␣
↪every polygon])
```

Note: Some polygons might not be tested (for example when a zone is found to be the closest already). To prevent this use `force_evaluation=True`.

A single timezone might be represented by multiple polygons and the distance to each of the candidate polygons is being computed and returned. Hence one may get multiple results for one timezone. Example:

```
longitude = 42.1052479
latitude = -16.622686
tf.closest_timezone_at(lng=longitude, lat=latitude, delta_degree=2,
                                    exact_computation=True, return_distances=True,␣
↪force_evaluation=True)
'''
returns ('uninhabited',
[80.66907784731714, 217.10924866254518, 293.5467252349301, 304.5274937839159, 238.
↪18462606485667, 267.918674688949, 207.43831938964408, 209.6790144988553, 228.
↪42135641542546],
['uninhabited', 'Indian/Antananarivo', 'Indian/Antananarivo', 'Indian/Antananarivo',
↪'Africa/Maputo', 'Africa/Maputo', 'Africa/Maputo', 'Africa/Maputo', 'Africa/Maputo
↪'])
'''
```

2.5 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)
```

2.6 Using vectorized input

Check numpy.vectorize and pandas.DataFrame.apply

2.7 Calling timezonefinder from the command line

Syntax:

```
python timezonefinder.py [-h] [-v] [-f {0,1}] lng lat
```

With `-v` you get verbose output, without it only the timezone name is being printed. Choose between functions `0: timezone_at()` and `1: certain_timezone_at()` with flag `-f` (default: timezone_at()). Please note that this is much slower than keeping a `TimezoneFinder` class directly in Python, because here all binary files are being opened again for each query.

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 most common timezone in that area.

TimezoneFinderL only offers the function `timezone_at()` (*API documentation*).

```
from timezonefinder import TimezoneFinderL

tf = TimezoneFinderL(in_memory=True)
latitude, longitude = 52.5061, 13.358
tf.timezone_at(lng=longitude, lat=latitude) # returns 'Europe/Berlin'
```

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

CHAPTER 3

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()
```

(continues on next page)

(continued from previous page)

```
    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


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

also see the pytz Doc.

3.3 Django

Maximising the chances of getting a result 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)
        if timezone_name is None:
            timezone_name = tf.closest_timezone_at(lng=lng, lat=lat)
            # maybe even increase the search radius when it is still None
    except ValueError:
        # the coordinates were out of bounds
        pass # {handle error}
    # ... 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/timezonefinder/file_converter.py [-inp /path/to/input.
↪json] [-out /path/to/output_folder]
```

Note: this script requires python3.6+ (as timezonefinder in general)

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 the `timezone-boundary-builder` version with ocean timezones.

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

CHAPTER 4

About

timezonefinder is a fast and lightweight 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 simple point in polygon tests. A few tweaks help to keep the computational requirements low and make this package fast. For example precomputed, so called "shortcuts" reduce the amount of timezone polygons to be checked (a kind of index for the polygons). See the documentation of the code itself for further explanation.

Current **data set** in use: precompiled timezone-boundary-builder (without oceans, (geo)JSON)

Note: The timezone polygons do NOT follow the shorelines. This makes the results of `closest_timezone_at()` and `certain_timezone_at()` somewhat meaningless.

Also see:

GitHub

PyPI

online GUI and API

conda-forge feedstock

ruby port: timezone_finder

download stats

4.1 License

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

4.2 Speed Test Results

obtained on MacBook Pro (15-inch, 2017), 2,8 GHz Intel Core i7

```
Speed Tests:
-------------
"realistic points": points included in a timezone

testing class <class 'timezonefinder.timezonefinder.TimezoneFinder'>

in memory mode: True
Numba: OFF (JIT compiled functions NOT in use)

testing 1000 realistic points
total time: 7.0352s
avg. points per second: 1.4 * 10^2

testing 1000 random points
total time: 3.1339s
avg. points per second: 3.2 * 10^2


in memory mode: False
Numba: ON (JIT compiled functions in use)

startup time: 0.001301s

testing 100000 realistic points
total time: 5.0705s
avg. points per second: 2.0 * 10^4

testing 100000 random points
total time: 3.2575s
avg. points per second: 3.1 * 10^4


in memory mode: True
Numba: ON (JIT compiled functions in use)

startup time: 0.03545s

testing 100000 realistic points
total time: 2.0659s
avg. points per second: 4.8 * 10^4
```

(continues on next page)

(continued from previous page)

```
testing 100000 random points
total time: 1.1928s
avg. points per second: 8.4 * 10^4


testing class <class 'timezonefinder.timezonefinder.TimezoneFinderL'>

startup time: 0.0005124s

using_numba()==True (JIT compiled functions in use)
in_memory=True

testing 100000 realistic points
total time: 0.1855s
avg. points per second: 5.4 * 10^5

testing 100000 random points
total time: 0.1722s
avg. points per second: 5.8 * 10^5

in_memory=False

testing 100000 realistic points
total time: 0.502s
avg. points per second: 2.0 * 10^5

testing 100000 random points
total time: 0.5323s
avg. points per second: 1.9 * 10^5
```

4.3 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
- available proximity algorithm `closest_timezone_at()`
- function `get_geometry()` enables querying timezones for their geometric shape (= multipolygon with holes)
- further speedup possible by the use of `numba` (code JIT compilation)

```
Startup times:
tzwhere: 0:00:29.365294
timezonefinder: 0:00:00.000888
33068.02 times faster

all other cross tests are not meaningful because tz_where is still using the outdated␣
↪tz_world data set
```

4.4 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*

4.5 Acknowledgements

Thanks to:

Adam for adding organisational features to the project and for helping me with publishing and testing routines.

snowman2 for creating the conda-forge recipe.

synapticarbors for fixing Numba import with py27.

zedrdave for valuable feedback.

CHAPTER 5

API documentation

5.1 TimezoneFinderL

class timezonefinder.**TimezoneFinderL**(*bin_file_location: Optional[str] = None*, *in_memory: bool = False*)

Bases: timezonefinder.timezonefinder.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

__init__(*bin_file_location: Optional[str] = None*, *in_memory: bool = False*)

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

bin_file_location

binary_data_attributes = ['shortcuts_direct_id']

in_memory

shortcuts_direct_id

timezone_at(*, *lng: float*, *lat: float*) → Optional[str]

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

timezone_names

static using_numba()
tests if Numba is being used or not

Returns True if the import of the JIT compiled algorithms worked. False otherwise

5.2 TimezoneFinder

class timezonefinder.**TimezoneFinder**(*bin_file_location: Optional[str] = None*, *in_memory: bool = False*)
Bases: timezonefinder.timezonefinder.AbstractTimezoneFinder

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

Opens the required timezone polygon data in binary files to enable fast access. Currently per half degree of latitude and per degree of longitude the set of all candidate polygons are stored. Because of these so called 'shortcuts' not all timezone polygons have to be tested during a query. 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

__init__(*bin_file_location: Optional[str] = None*, *in_memory: bool = False*)
Initialize self. See help(type(self)) for accurate signature.

bin_file_location

binary_data_attributes = ['poly_zone_ids', 'poly_coord_amount', 'poly_adr2data', 'poly

certain_timezone_at(*, *lng: float*, *lat: float*) → Optional[str]
looks up in which polygon the point certainly is included in

Note: this is much slower 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

closest_timezone_at(*, *lng: float*, *lat: float*, *delta_degree: int = 1*, *exact_computation: bool = False*, *return_distances: bool = False*, *force_evaluation: bool = False*)
Searches for the closest polygon in the surrounding shortcuts

Computes the (approximate) distance to all the polygons within delta_degree degree lng and lat Make sure that the point does not lie within a polygon

Note: the algorithm won't find the closest polygon when it's on the 'other end of earth' (it can't search beyond the 180 deg lng border!)

Note: x degrees lat are not the same distance apart than x degree lng! This is also the reason why there could still be a closer polygon even though you got a result already. In order to make sure to get the closest polygon, you should increase the search radius until you get a result and then increase it once more (and take that result). This should however only make a difference in rare cases.

Parameters

- **lng** – longitude in degree
- **lat** – latitude in degree
- **delta_degree** – the 'search radius' in degree. determines the polygons to be checked (shortcuts to include)
- **exact_computation** – when enabled the distance to every polygon edge is computed (=computationally more expensive!), instead of only evaluating the distances to all the vertices. NOTE: This only makes a real difference when polygons are very close.
- **return_distances** – when enabled the output looks like this: `( 'tz_name_of_the_closest_polygon',[ distances to all polygons in km], [tz_names of all polygons])`
- **force_evaluation** – whether all distances should be computed in any case

Returns the timezone name of the closest found polygon, the list of distances or None

compile_id_list (*polygon_id_list*, *nr_of_polygons*)
sorts the polygons_id list from least to most occurrences of the zone ids (->speed up)

only 4.8% of all shortcuts include polygons from more than one zone but only for about 0.4% sorting would be beneficial (zones have different frequencies) in most of those cases there are only two types of zones (= entries in counted_zones) and one of them has only one entry. the polygon lists of all single shortcut are already sorted (during compilation of the binary files) sorting should be used for closest_timezone_at(), because only in that use case the polygon lists are quite long (multiple shortcuts are being checked simultaneously).

Parameters

- **polygon_id_list** – input list of polygon
- **nr_of_polygons** – length of polygon_id_list

Returns sorted list of polygon_ids, sorted list of zone_ids, boolean: do all entries belong to the same zone

coords_of (*polygon_nr: int = 0*)

get_geometry (*tz_name=''*, *tz_id=0*, *use_id=False*, *coords_as_pairs=False*)
retrieves the geometry of a timezone polygon

Parameters

- **tz_name** – one of the names in `getattr(self, TIMEZONE_NAMES)`
- **tz_id** – the id of the timezone (=index in `getattr(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 (*polygon_nr: int*, *coords_as_pairs: bool = False*)

hole_adr2data

hole_coord_amount

hole_data

hole_registry

id_list (*polygon_id_list*, *nr_of_polygons*)

Parameters

- **polygon_id_list** –
- **nr_of_polygons** – length of polygon_id_list

Returns (list of zone_ids, boolean: do all entries belong to the same zone)

id_of (*polygon_nr: int = 0*)

ids_of (*iterable*)

in_memory

poly_adr2data

poly_coord_amount

poly_data

poly_max_values

poly_nr2zone_id

poly_zone_ids

polygon_ids_of_shortcut (*x: int = 0*, *y: int = 0*)

shortcuts_adr2data

shortcuts_data

shortcuts_entry_amount

shortcuts_unique_id

timezone_at (*, *lng: float*, *lat: float*) → Optional[str]

looks up in which timezone the given coordinate is possibly included in

to speed things up there are shortcuts being used (stored in a binary file) especially for large polygons it is expensive to check if a point is really included, so certain simplifications are made and even when you get a hit the point might actually not be inside the polygon (for example when there is only one timezone nearby) if you want to make sure a point is really inside a timezone use `certain_timezone_at()`

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

timezone_names

static using_numba()
tests if Numba is being used or not

Returns True if the import of the JIT compiled algorithms worked. False otherwise

CHAPTER 6

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:

6.1 Types of Contributions

6.1.1 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.).

6.1.2 Fix Bugs

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

6.1.3 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.

6.1.4 Write Documentation

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

6.1.5 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 :)

6.2 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 7

Changelog

7.1 5.0.0 (TBA)

- TODO removed data files
- **TODO added "extra" simplifying the installation of TODO external data** TODO document! also in minimal example in readme!
- TODO specify minimal version of data extra! ('pin')
- TODO describe loading priority
- TODO private repo, without the actual data. for data files! but regular upload to pypI,
- TODO importlib_resources dependency

TODO document class attributes TODO create variables for used dtype for each type of data (polygon address, coordinate...) more "intelligent" binary file creation settings: name, dtype etc. combined

7.2 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

7.3 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)

7.4 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

7.5 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

7.6 4.1.0 (2019-07-07)

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

7.7 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

7.8 4.0.2 (2019-04-01)

- updated the data to 2019a

7.9 4.0.1 (2019-03-12)

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

7.10 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)

7.11 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

7.12 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)

7.13 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".

7.14 3.3.0 (2018-11-17)

- updated the data to 2018g

7.15 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()

7.16 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

7.17 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)

7.18 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

7.19 3.0.1 (2018-05-30)

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

7.20 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

7.21 2.1.2 (2017-11-20)

- bugfix: possibly uninitialized variable in closest_timezone_at()

7.22 2.1.1 (2017-11-20)

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

7.23 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)

7.24 2.0.1 (2017-04-08)

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

7.25 2.0.0 (2017-04-07)

- **ATTENTION: major change!: there is a second version of timezonefinder now: timezonefinderL. There the data has been** 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

7.26 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)

7.27 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)

7.28 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

7.29 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

7.30 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)

7.31 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

7.32 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

7.33 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

7.34 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 int**
  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 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

Symbols

B

C

G

H

I

P

S

T

U