backports.zoneinfo

Paul Ganssle

May 25, 2021

CONTENTS

This was originally the reference implementation for **PEP 615**, which adds support for the IANA time zone database to the Python standard library, but now serves as a backport of the module to Python 3.6+ (including PyPy).

The upstream documentation can be found at `zoneinfo`. A mirror of the documentation pinned to the version supported in the backport can be found at *`backports.zoneinfo`*.

This is the documentation for version 0.2.1.

CHAPTER

ONE

DOCUMENTATION

Contents:

1.1 backports.zoneinfo — IANA time zone support

The *zoneinfo* module provides a concrete time zone implementation to support the IANA time zone database as originally specified in **PEP 615**. By default, *zoneinfo* uses the system's time zone data if available; if no system time zone data is available, the library will fall back to using the first-party tzdata package available on PyPI.

See also:

Module: zoneinfo The standard library module zoneinfo, of which this is a backport.

Module: datetime Provides the time and datetime types with which the *ZoneInfo* class is designed to be used.

Package tzdata First-party package maintained by the CPython core developers to supply time zone data via PyPI.

1.1.1 Using ZoneInfo

ZoneInfo is a concrete implementation of the datetime.tzinfo abstract base class, and is intended to be attached to tzinfo, either via the constructor, the datetime.replace method or datetime.astimezone:

```
>>> from backports.zoneinfo import ZoneInfo
>>> from datetime import datetime, timedelta

>>> dt = datetime(2020, 10, 31, 12, tzinfo=ZoneInfo("America/Los_Angeles"))
>>> print(dt)
2020-10-31 12:00:00-07:00

>>> dt.tzname()
'PDT'
```

Datetimes constructed in this way are compatible with datetime arithmetic and handle daylight saving time transitions with no further intervention:

```
>>> dt_add = dt + timedelta(days=1)

>>> print(dt_add)
2020-11-01 12:00:00-08:00
```

(continues on next page)

(continued from previous page)

```
>>> dt_add.tzname()
'PST'
```

These time zones also support the `fold` attribute introduced in **PEP 495**. During offset transitions which induce ambiguous times (such as a daylight saving time to standard time transition), the offset from *before* the transition is used when `fold=0`, and the offset *after* the transition is used when `fold=1`, for example:

```
>>> dt = datetime(2020, 11, 1, 1, tzinfo=ZoneInfo("America/Los_Angeles"))
>>> print(dt)
2020-11-01 01:00:00-07:00

>>> print(dt.replace(fold=1))
2020-11-01 01:00:00-08:00
```

When converting from another time zone, the fold will be set to the correct value:

```
>>> from datetime import timezone
>>> LOS_ANGELES = ZoneInfo("America/Los_Angeles")
>>> dt_utc = datetime(2020, 11, 1, 8, tzinfo=timezone.utc)

>>> # Before the PDT -> PST transition
>>> print(dt_utc.astimezone(LOS_ANGELES))
2020-11-01 01:00:00-07:00

>>> # After the PDT -> PST transition
>>> print((dt_utc + timedelta(hours=1)).astimezone(LOS_ANGELES))
2020-11-01 01:00:00-08:00
```

1.1.2 Data sources

The `zoneinfo` module does not directly provide time zone data, and instead pulls time zone information from the system time zone database or the first-party PyPI package tzdata, if available. Some systems, including notably Windows systems, do not have an IANA database available, and so for projects targeting cross-platform compatibility that require time zone data, it is recommended to declare a dependency on tzdata. If neither system data nor tzdata are available, all calls to *ZoneInfo* will raise *ZoneInfoNotFoundError*.

Configuring the data sources

When `ZoneInfo(key)` is called, the constructor first searches the directories specified in *TZPATH* for a file matching `key`, and on failure looks for a match in the tzdata package. This behavior can be configured in three ways:

1. The default *TZPATH* when not otherwise specified can be configured at *compile time*.
2. *TZPATH* can be configured using *an environment variable*.
3. At *runtime*, the search path can be manipulated using the *reset_tzpath()* function.

Compile-time configuration

The default *TZPATH* includes several common deployment locations for the time zone database (except on Windows, where there are no "well-known" locations for time zone data). On POSIX systems, downstream distributors and those building Python from source who know where their system time zone data is deployed may change the default time zone path by specifying the compile-time option `TZPATH` (or, more likely, the `configure` flag `--with-tzpath`), which should be a string delimited by `os.pathsep`.

On all platforms, the configured value is available as the `TZPATH` key in `sysconfig.get_config_var()`.

Note: This option is currently not available in the backport.

Environment configuration

When initializing *TZPATH* (either at import time or whenever *reset_tzpath()* is called with no arguments), the `zoneinfo` module will use the environment variable `PYTHONTZPATH`, if it exists, to set the search path.

PYTHONTZPATH

This is an `os.pathsep`-separated string containing the time zone search path to use. It must consist of only absolute rather than relative paths. Relative components specified in `PYTHONTZPATH` will not be used, but otherwise the behavior when a relative path is specified is implementation-defined; CPython will raise *InvalidTZPathWarning*, but other implementations are free to silently ignore the erroneous component or raise an exception.

To set the system to ignore the system data and use the tzdata package instead, set `PYTHONTZPATH=""`.

Runtime configuration

The TZ search path can also be configured at runtime using the *reset_tzpath()* function. This is generally not an advisable operation, though it is reasonable to use it in test functions that require the use of a specific time zone path (or require disabling access to the system time zones).

1.1.3 The `ZoneInfo` class

class backports.zoneinfo.**ZoneInfo**(*key*)

A concrete `datetime.tzinfo` subclass that represents an IANA time zone specified by the string `key`. Calls to the primary constructor will always return objects that compare identically; put another way, barring cache invalidation via *ZoneInfo.clear_cache()*, for all values of `key`, the following assertion will always be true:

```
a = ZoneInfo(key)
b = ZoneInfo(key)
assert a is b
```

`key` must be in the form of a relative, normalized POSIX path, with no up-level references. The constructor will raise `ValueError` if a non-conforming key is passed.

If no file matching `key` is found, the constructor will raise *ZoneInfoNotFoundError*.

The `ZoneInfo` class has two alternate constructors:

classmethod ZoneInfo.**from_file**(*fobj*, */*, *key=None*)
Constructs a `ZoneInfo` object from a file-like object returning bytes (e.g. a file opened in binary mode or an `io.BytesIO` object). Unlike the primary constructor, this always constructs a new object.

The `key` parameter sets the name of the zone for the purposes of `__str__()` and `__repr__()`.

Objects created via this constructor cannot be pickled (see *pickling*).

classmethod ZoneInfo.**no_cache**(*key*)
An alternate constructor that bypasses the constructor's cache. It is identical to the primary constructor, but returns a new object on each call. This is most likely to be useful for testing or demonstration purposes, but it can also be used to create a system with a different cache invalidation strategy.

Objects created via this constructor will also bypass the cache of a deserializing process when unpickled.

> **Caution:** Using this constructor may change the semantics of your datetimes in surprising ways, only use it if you know that you need to.

The following class methods are also available:

classmethod ZoneInfo.**clear_cache**(*, *only_keys=None*)
A method for invalidating the cache on the `ZoneInfo` class. If no arguments are passed, all caches are invalidated and the next call to the primary constructor for each key will return a new instance.

If an iterable of key names is passed to the `only_keys` parameter, only the specified keys will be removed from the cache. Keys passed to `only_keys` but not found in the cache are ignored.

> **Warning:** Invoking this function may change the semantics of datetimes using `ZoneInfo` in surprising ways; this modifies process-wide global state and thus may have wide-ranging effects. Only use it if you know that you need to.

The class has one attribute:

ZoneInfo.**key**
This is a read-only attribute that returns the value of `key` passed to the constructor, which should be a lookup key in the IANA time zone database (e.g. `America/New_York`, `Europe/Paris` or `Asia/Tokyo`).

For zones constructed from file without specifying a `key` parameter, this will be set to `None`.

> **Note:** Although it is a somewhat common practice to expose these to end users, these values are designed to be primary keys for representing the relevant zones and not necessarily user-facing elements. Projects like CLDR (the Unicode Common Locale Data Repository) can be used to get more user-friendly strings from these keys.

String representations

The string representation returned when calling `str` on a *`ZoneInfo`* object defaults to using the *`ZoneInfo.key`* attribute (see the note on usage in the attribute documentation):

```
>>> zone = ZoneInfo("Pacific/Kwajalein")
>>> str(zone)
'Pacific/Kwajalein'
```

(continues on next page)

(continued from previous page)

```
>>> dt = datetime(2020, 4, 1, 3, 15, tzinfo=zone)
>>> f"{dt.isoformat()} [{dt.tzinfo}]"
'2020-04-01T03:15:00+12:00 [Pacific/Kwajalein]'
```

For objects constructed from a file without specifying a `key` parameter, `str` falls back to calling `repr()`. `ZoneInfo`'s `repr` is implementation-defined and not necessarily stable between versions, but it is guaranteed not to be a valid `ZoneInfo` key.

Pickle serialization

Rather than serializing all transition data, `ZoneInfo` objects are serialized by key, and `ZoneInfo` objects constructed from files (even those with a value for `key` specified) cannot be pickled.

The behavior of a `ZoneInfo` file depends on how it was constructed:

1. `ZoneInfo(key)`: When constructed with the primary constructor, a `ZoneInfo` object is serialized by key, and when deserialized, the deserializing process uses the primary and thus it is expected that these are expected to be the same object as other references to the same time zone. For example, if `europe_berlin_pkl` is a string containing a pickle constructed from `ZoneInfo("Europe/Berlin")`, one would expect the following behavior:

   ```
   >>> a = ZoneInfo("Europe/Berlin")
   >>> b = pickle.loads(europe_berlin_pkl)
   >>> a is b
   True
   ```

2. `ZoneInfo.no_cache(key)`: When constructed from the cache-bypassing constructor, the `ZoneInfo` object is also serialized by key, but when deserialized, the deserializing process uses the cache bypassing constructor. If `europe_berlin_pkl_nc` is a string containing a pickle constructed from `ZoneInfo.no_cache("Europe/Berlin")`, one would expect the following behavior:

   ```
   >>> a = ZoneInfo("Europe/Berlin")
   >>> b = pickle.loads(europe_berlin_pkl_nc)
   >>> a is b
   False
   ```

3. `ZoneInfo.from_file(fobj, /, key=None)`: When constructed from a file, the `ZoneInfo` object raises an exception on pickling. If an end user wants to pickle a `ZoneInfo` constructed from a file, it is recommended that they use a wrapper type or a custom serialization function: either serializing by key or storing the contents of the file object and serializing that.

This method of serialization requires that the time zone data for the required key be available on both the serializing and deserializing side, similar to the way that references to classes and functions are expected to exist in both the serializing and deserializing environments. It also means that no guarantees are made about the consistency of results when unpickling a `ZoneInfo` pickled in an environment with a different version of the time zone data.

1.1.4 Functions

backports.zoneinfo.**available_timezones**()

Get a set containing all the valid keys for IANA time zones available anywhere on the time zone path. This is recalculated on every call to the function.

This function only includes canonical zone names and does not include "special" zones such as those under the `posix/` and `right/` directories, or the `posixrules` zone.

> **Caution:** This function may open a large number of files, as the best way to determine if a file on the time zone path is a valid time zone is to read the "magic string" at the beginning.

Note: These values are not designed to be exposed to end-users; for user facing elements, applications should use something like CLDR (the Unicode Common Locale Data Repository) to get more user-friendly strings. See also the cautionary note on *ZoneInfo.key*.

---

backports.zoneinfo.**reset_tzpath**(*to=None*)

Sets or resets the time zone search path (*TZPATH*) for the module. When called with no arguments, *TZPATH* is set to the default value.

Calling `reset_tzpath` will not invalidate the *ZoneInfo* cache, and so calls to the primary `ZoneInfo` constructor will only use the new `TZPATH` in the case of a cache miss.

The `to` parameter must be a sequence of strings or `os.PathLike` and not a string, all of which must be absolute paths. `ValueError` will be raised if something other than an absolute path is passed.

1.1.5 Globals

backports.zoneinfo.**TZPATH**

A read-only sequence representing the time zone search path – when constructing a `ZoneInfo` from a key, the key is joined to each entry in the `TZPATH`, and the first file found is used.

`TZPATH` may contain only absolute paths, never relative paths, regardless of how it is configured.

The object that `zoneinfo.TZPATH` points to may change in response to a call to *reset_tzpath()*, so it is recommended to use `zoneinfo.TZPATH` rather than importing `TZPATH` from `zoneinfo` or assigning a long-lived variable to `zoneinfo.TZPATH`.

For more information on configuring the time zone search path, see *Configuring the data sources*.

1.1.6 Exceptions and warnings

exception backports.zoneinfo.**ZoneInfoNotFoundError**

Raised when construction of a *ZoneInfo* object fails because the specified key could not be found on the system. This is a subclass of `KeyError`.

exception backports.zoneinfo.**InvalidTZPathWarning**

Raised when *PYTHONTZPATH* contains an invalid component that will be filtered out, such as a relative path.

1.2 Changelog

1.2.1 Version 0.2.1 (2020-06-18)

- Fixed an issue where the C implementation of `ZoneInfo.__init_subclass__` was not a classmethod, causing errors when attempting to subclass `ZoneInfo` (GH-82, GH-83).

1.2.2 Version 0.2.0 (2020-05-29)

- Added support for PyPy 3.6 (GH-74); when installed on PyPy, the library will not use the C extension, since benchmarks indicate that the pure Python implementation is faster.

1.2.3 Version 0.1.0 (2020-05-26)

This is the first public release of `backports.zoneinfo`. It contains all the features from the `zoneinfo` release in Python 3.9.0b1, with the following changes:

- Added support for Python 3.6, 3.7 and 3.8 (GH-69, GH-70).
- The module is in the `backports` namespace rather than `zoneinfo`.
- There is no support for compile-time configuration of `TZPATH`.
- Fixed use-after-free in the `module_free` function (bpo-40705, GH-69).
- Minor refactoring to the C extension to avoid compiler warnings (bpo-40686, bpo-40714, CPython PR #20342, GH-72).
- Removed unused imports, unused variables and other minor de-linting (GH-71).

1.3 Maintainer's Guide

Although this was the original implementation of the `zoneinfo` module, after Python 3.9, it is now a backport, and to the extent that there is a "canonical" repository, the CPython repository has a stronger claim than this one. Accepting outside PRs against this repository is difficult because we are not set up to collect CLAs for CPython. It is easier to accept PRs against CPython and import them here if possible.

The code layout is very different between the two, and unfortunately (partially because of the different layouts, and the different module names), the code has diverged, so keeping the two in sync is not as simple as copy-pasting one into the other. For now, the two will need to be kept in sync manually.

1.3.1 Development environment

Maintenance scripts, releases, and tests are orchestrated using `tox` environments to manage the requirements of each script. The details of each environment can be found in the `tox.ini` file in the repository root.

The repository also has pre-commit configured to automatically enforce various code formatting rules on commit. To use it, install pre-commit and run `pre-commit install` in the repository root to install the git commit hooks.

1.3.2 Making a release

Releases are automated via the `build-release.yml` GitHub Actions workflow. The project is built on every push; whenever a *tag* is pushed, the build artifacts are released to Test PyPI, and when a GitHub release is made, the project is built and released to PyPI (this is a workaround for the lack of "draft releases" on PyPI, and the two actions can be unified when that feature is added).

To make a release:

1. Update the version number in `src/backports/zoneinfo/_version.py` and make a PR (if you want to be cautious, start with a `.devN` release intended only for PyPI).
2. Tag the repository with the current version – you can use the `scripts/tag_release.sh` script in the repository root to source the version automatically from the current module version.
3. Push the tag to GitHub (e.g. `git push upstream 0.1.0.dev0`). This will trigger a release to Test PyPI. The PR does not need to be merged at this point if you are only planning to release to TestPyPI, but any "test only" tags should be deleted when the process is complete.
4. Wait for the GitHub action to succeed, then check the results on https://test.pypi.org/project/backports.zoneinfo .
5. If everything looks good, go into the GitHub repository's "releases" tab and click "Draft a new release"; type the name of the tag into the box, fill out the remainder of the form, and click "Publish". (Only do this step for non-dev releases).
6. Check that the release action has succeeded, then check that everything looks OK on https://pypi.org/project/backports.zoneinfo/ .

If there's a problem with the release, make a post release by appending `.postN` to the current version, e.g. `0.1.1` → `0.1.1.post0`. If the problem is sufficiently serious, yank the broken version.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX