PyStandardPaths Documentation

Release 0.3.2

Tzu-ping Chung

Mar 24, 2018

Contents

PyStandardPaths provides methods for accessing standard paths.

This module contains functions to query standard locations on the local filesystem, for common tasks such as user-specific directories or system-wide configuration directories. The functions, location names, and the implementation of how paths are calculated, are based on Qt 5's QStandardPaths class.

The current implementation matches that of Qt 5.4.1. This matching-Qt-version information can be obtained via `standardpaths.QTVERSION` (as a tuple of integers) and `standardpaths.__qtversion__` (as a string).

CHAPTER 1

Installation

PyStandardPaths can be installed with `pip`:

```
pip install pystandardpaths
```

Currently OS X, Windows, and Unix systems conforming to freedesktop.org specifications are supported.

Implementations on OS X and Windows are based on `ctypes`, and dependencies vary between operating systems and Python versions.

CHAPTER 2

Contents

2.1 Usage

2.1.1 Basic Use

All functionalities of PyStandardPaths can be accessed by

```
import standardpaths
```

The first thing you need to do is to tell PyStandardPaths about your application:

```
standardpaths.configure(application_name='Pepsi', organization_name='Tzu-ping Chung')
```

Both application and organization names can be left empty, which is useful if you want to configure PyStandardPaths to access "organization-wide" settings that can be shared between multiple applications of your organization.

To get a path that you can write things into, you can use *get_writable_path()*. For example, to get the current user's Desktop:

```
path = standardpaths.get_writable_path('desktop')
```

This will return a `pathlib.Path` object, pointing to the desktop directory, e.g. `/Users/uranusjr/Desktop` on my Mac.

To get a list of paths to look for files in a system, use *get_standard_paths()*. Taking my Mac as an example again, this prints a list of cache directories for my Pepsi application:

```
>>> for path in standardpaths.get_standard_paths('fonts'):
...     print(str(path))
...
/Users/uranusjr/Library/Caches/Tzu-ping Chung/Pepsi
/Library/Caches/Tzu-ping Chung/Pepsi
```

For a complete list of locations you can use, refer to the documentation of *Location*.

2.1.2 Advanced Use

Both *get_writable_path()* and *get_standard_paths()* take an optional second argument `config` that overrides the global configuration. Useful for accessing directories owned by *another* application:

```
>>> config = standardpaths.Config(organization_name='M05', application_name='<Y')
>>> print(standardpaths.get_writable_path('cache', config=config))
PosixPath('/home/uranusjr/.cache/M05/<Y')
```

Instead of using strings, you can also use a *Location* `enum` value as the first argument:

```
path = standardpaths.get_writable_path(standardpaths.Location.applications)
```

2.2 API References

Documentation in this section is copied from documentation of QStandardPaths with minimal modification.

`standardpaths.`**`configure`**(*application_name=''*, *organization_name=''*)
Configure default application information used by PyStandardPaths.

See also:

get_config() and *Config*.

`standardpaths.`**`get_config`**()
Get the current configuration of application information.

Return type *Config*

`standardpaths.`**`get_writable_path`**(*location*, *config=None*)
Get the directory where files of type should be written to. A *LocationError* is raised if the location cannot be determined.

Return type `pathlib.Path`

Note: The storage location returned can be a directory that does not exist; i.e., it may need to be created by the system or the user.

`standardpaths.`**`get_standard_paths`**(*location*, *config=None*)
Get all the directories where files of type belong.

The list of directories is sorted from high to low priority, starting with *get_writable_path()* if it can be determined. This list is empty if no locations for type are defined.

Return type `pathlib.Path`

See also:

get_writable_path().

`class` `standardpaths.`**`Config`**(*application_name=''*, *organization_name=''*)
Configuration class that holds application information.

See also:

configure() and *get_config()*.

class standardpaths.**LocationError**
Bases: OSError

Exception class raised to indicate an error during path resolution.

class standardpaths.**Location**
Bases: enum.Enum

Describe the different locations that can be queried using functions such as *get_writable_path()* and *get_standard_paths()*.

Some of the values in this enum represent a user configuration. Such enum values will return the same paths in different applications, so they could be used to share data with other applications. Other values are specific to this application. Each enum value in the table below describes whether it's application-specific or generic.

Application-specific directories should be assumed to be unreachable by other applications. Therefore, files placed there might not be readable by other applications, even if run by the same user. On the other hand, generic directories should be assumed to be accessible by all applications run by this user, but should still be assumed to be unreachable by applications by other users.

Data interchange with other users is out of the scope of PyStandardPaths.

app_data = 17
A directory location where persistent application data can be stored. This is an application-specific directory. To obtain a path to store data to be shared with other applications, use *generic_data*. The returned path is never empty. On the Windows operating system, this returns the roaming path.

app_local_data = None
The local settings path on the Windows operating system. On all other platforms, it returns the same value as *app_data*.

applications = 3
The directory containing the user applications (either executables, application bundles, or shortcuts to them). This is a generic value. Note that installing applications may require additional, platform-specific operations. Files, folders or shortcuts in this directory are platform-specific.

cache = 10
A directory location where user-specific non-essential (cached) data should be written. This is an application-specific directory. The returned path is never empty.

config = 13
A directory location where user-specific configuration files should be written. This may be either a generic value or application-specific, and the returned path is never empty.

data = 9
The same value as *app_local_data*. This enumeration value is deprecated. Using *app_data* is preferable since on Windows, the roaming path is recommended.

desktop = 0
The user's desktop directory. This is a generic value.

documents = 1
The directory containing user document files. This is a generic value. The returned path is never empty.

download = 14
A directory for user's downloaded files. This is a generic value. If no directory specific for downloads exists, a sensible fallback for storing user documents is returned.

fonts = 2
The directory containing user's fonts. This is a generic value. Note that installing fonts may require additional, platform-specific operations.

generic_cache = 15
A directory location where user-specific non-essential (cached) data, shared across applications, should be written. This is a generic value. Note that the returned path may be empty if the system has no concept of shared cache.

generic_config = 16
A directory location where user-specific configuration files shared between multiple applications should be written. This is a generic value and the returned path is never empty.

generic_data = 11
A directory location where persistent data shared across applications can be stored. This is a generic value. The returned path is never empty.

home = 8
The user's home directory (the same as `os.path.expanduser('~')`). On Unix systems, this is equal to the `HOME` environment variable. This value might be generic or application-specific, but the returned path is never empty.

log = 'log'
A directory location where user-specific log files should be written. This is an application-specific value. The returned path is never empty.

movies = 5
The directory containing the user's movies and videos. This is a generic value. If no directory specific for movie files exists, a sensible fallback for storing user documents is returned.

music = 4
The directory containing the user's music or other audio files. This is a generic value. If no directory specific for music files exists, a sensible fallback for storing user documents is returned.

pictures = 6
The directory containing the user's pictures or photos. This is a generic value. If no directory specific for picture files exists, a sensible fallback for storing user documents is returned.

runtime = 12
A directory location where runtime communication files should be written, like Unix local sockets. This is a generic value. The returned path may be empty on some systems.

temp = 7
A directory where temporary files can be stored (the same as `tempfile.gettempdir()`). The returned value might be application-specific, shared among other applications for this user, or even system-wide. The returned path is never empty.

2.3 Contributing

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

You can contribute in many ways:

2.3.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/uranusjr/pystandardpaths/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

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

Implement Features

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

Write Documentation

PyStandardPaths could always use more documentation, whether as part of the official PyStandardPaths docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/uranusjr/pystandardpaths/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.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

2.3.2 Get Started!

Ready to contribute? Here's how to set up `standardpaths` for local development.

1. Fork the `standardpaths` repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/pystandardpaths.git
```

3. Install your local copy into a virtualenv. Assuming you have Python 3.4 with `venv`, this is how you set up your fork for local development:

```
$ python3 -m venv venv/pystandardpaths
$ . venv/pystandardpaths
$ cd pystandardpaths/
$ python setup.py develop
$ pip install -r requirements.txt
```

4. Create a branch for local development:

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

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 standardpaths tests
$ python setup.py nosetests
$ tox
```

6. 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
```

7. Submit a pull request through the GitHub website.

2.3.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.

2.3.4 Tips

To run a subset of tests:

```
$ nosetests tests.test_basic:PyStandardPathsTests.test_get_writable_path
```

2.4 Credits

2.4.1 Contributors

- Tzu-ping Chung <uranusjr@gmail.com>

2.5 History

2.5.1 0.3.2 (2018-03-24)

- Better compatibility to modern pip versions.

2.5.2 0.3.1 (2015-07-24)

- Fix platform-dependent loading on Python 2.

2.5.3 0.3.0 (2015-05-05)

- Always raise LocationError on resolution error.
- Packaging problem fixed.

2.5.4 0.2.0 (2015-05-02)

- Add log location support.

2.5.5 0.1.0 (2015-05-02)

- First release on PyPI.

Index