pymenu Documentation

Release

Charles Bouchard-Légaré

Jun 07, 2017

Contents

An API for menu definitions.

Warning: Before 1.0 release, this project will not follow any reliable versioning scheme. Do not expect backward-compatibility between versions!

Warning: This project is not stable at all! Parts of it might be moved to external packages without notice.

This project was intented to be used with the extensible dmenu wrapper as a menu API for Qtile.

pymenu is free software and licensed under the GNU Lesser General Public License v3.

CHAPTER 1

Features

- Simple python interfaces for menus
- Easy to configure using simple dictionaries and the filesystem
- Extension available for XDG-based menus (including launching applications defined in *desktop* files).

CHAPTER 2

Credits

This package was created with Cookiecutter and the cblegare/pythontemplate project template.

User manual

Installation

Stable release

To install *pymenu*, run this command in your terminal:

```
$ pip install pymenu
```

This is the preferred method to install *pymenu*, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

From sources

The sources for *pymenu* can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/cblegare/pymenu
```

Or download the tarball:

```
$ curl -OL https://github.com/cblegare/pymenu/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

Basic Usage

A menu from with *pymenu* is made of two things:

The root menu entry Menu entries are trees where each branch is a submenu and each leaf is an item.

Prompt *pymenu* needs to know how to display the menu to the user. This is what a **prompt** is used to.

A silly command line file manager

Given the following directory structure:

```
.
- demo.py
- folder
|   - deepfile
|   - subfolder
|       - deeperfile
- some_file

2 directories, 4 files
```

Here is a simple script that makes a file manager for browsing folder and printing the chosen file name.

```python
#!/usr/bin/python

from pymenu import Menu, FileSystemMenuEntry, SimpleCommandPrompt

menu_entry = FileSystemMenuEntry('.')
prompt = SimpleCommandPrompt()

my_menu = Menu(menu_entry, prompt)

choice = my_menu.choose_value()
print(choice)
```

You can see it in action in the following animated gif.

Leveraging *xdmenu*

pymenu provides a simple API for building menus. It leverages xdmenu (must be installed) for delegating the display to an implementation of dmenu.

Lets change the above silly example in order to browse our files using *dmenu*.

```python
#!/usr/bin/python

from pymenu import Menu, FileSystemMenuEntry
from pymenu.ext.xdmenu import DmenuPrompt

menu_entry = FileSystemMenuEntry('.')
prompt = DmenuPrompt()
```

```
my_menu = Menu(menu_entry, prompt)

choice = my_menu.choose_value()
print(choice)
```

Changes are emphasized. Simple enough, right?

Leveraging *XDG*

pymenu ships with an extension providing support for XDG menu definitons. This can be useful when using a simple window manager that is not XDG-compliant, such as Qtile and still wanting a XDG-based applications menu.

Here is a simple script for launching an application based on XDG menu definitons.

```
#!/usr/bin/python

from pymenu import Menu
from pymenu.ext.xdmenu import DmenuPrompt
from pymenu.ext.pyxdg import make_xdg_menu_entry, launch_xdg_menu_entry

menu_entry = make_xdg_menu_entry()
prompt = DmenuPrompt()

my_menu = Menu(menu_entry, prompt)

choice = my_menu.choose_value()
launch_xdg_menu_entry(choice)
```

Note: *pymenu* has all you need for launching default applications as per the XDG specification. Do not rely on this API, because it may (will) be moved to another package!

Performance Issues

pymenu do not implement lazy loading of menu entries. This means that a menu can use up a lot of RAM. Also, Creating a menu may take some time, especially when using XDG because of all the heavy XML files that needs parsing in the process.

Please help! See *Contributing* for more informations.

Project Information

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:

Types of Contributions

Report Bugs

Report bugs at https://github.com/cblegare/pymenu/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” and “help wanted” is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with “enhancement” and “help wanted” is open to whoever wants to implement it.

Write Documentation

pymenu could always use more documentation, whether as part of the official *pymenu* 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/cblegare/pymenu/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 :)

Get Started!

Ready to contribute? Here’s how to set up *pymenu* for local development.

1. Fork the *pymenu* repo on GitHub.
2. Clone your fork locally:

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

3. Install your local copy into a virtualenv. Assuming you have Python 3.5 installed, this is how you set up your fork for local development:

   ```
   $ python3 -m venv pymenu
   $ cd pymenu/
   $ bin/pip install --editable . # or bin/python setup.py develop
   ```

```
```

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:

```
$ python setup.py test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

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.

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.
3. The pull request should work for Python 2.7 and up. Check https://travis-ci.org/cblegare/pymenu/pull_requests and make sure that the tests pass for all supported Python versions.

Thanks :)

History

1.0 (2017-05-09)

- First release on PyPI.

License

Credits

Contributors

- Charles Bouchard-Légaré <cblegare.atl@ntis.ca>

Development resources

setup module

The *setup.py* file is a swiss knife for various tasks.

Start by creating a virtual python environment:

```
$ python -m venv .
```

You now can use this isolated clean python environment:

```
$ bin/python --version
Python 3.5.2
```

You may also activate it for the current shell. POSIX shells would use:

```
$ . bin/activate
```

running tests

We use py.test for running tests because it is amazing. Run it by invoking the simple *test* alias of *setup.py*:

```
$ bin/python setup.py test
```

This will also check codestyle and test coverage.

checking code style

We use flake8 for enforcing coding standards. Run it by invoking the simple *lint* alias of *setup.py*:

```
$ bin/python setup.py lint
```

building source distirbutions

Standard *sdist* is supported:

```
$ bin/python setup.py sdist
```

building binary distributions

Use the wheel distribution standard:

```
$ bin/python setup.py bdist_wheel
```

building html documentation

Use *setup.py* to build the documentation:

```
$ bin/python setup.py docs
```

A make implementation is not required on any platform, thanks to the `setup.Documentation` class.

class `setup.`**`Documentation`**(*dist*, ***kw*)

Make the documentation (without the *Make* program).

Note: This command will not allow any warning from Sphinx, treating them as errors.

Construct the command for dist, updating vars(self) with any keyword parameters.

cleaning your workspace

We also included a custom command which you can invoke through *setup.py*:

```
$ bin/python setup.py clean
```

The `setup.Clean` command is set to clean the following file patterns:

class `setup.`**`Clean`**(*dist*, ***kw*)

Custom clean command to tidy up the project.

Construct the command for dist, updating vars(self) with any keyword parameters.

`default_patterns` **= ['build', 'dist', '*.egg-info', '*.egg', '*.pyc', '*.pyo', '*~', '__pycache__', '.tox', '.coverage', 'html**

Automated tests

The `tests` package provides automated testing for *'pymenu'*.

Tests are known to assess software behavior and find bugs. They are also used as part of the code's documentation, as a design tool or for preventing regressions.

See also:

- http://stackoverflow.com/questions/4904096/whats-the-difference-between-unit-functional-acceptance-and-integration-test
- http://stackoverflow.com/questions/520064/what-is-unit-test-integration-test-smoke-test-regression-test

Unit tests

Exercise the smallest pieces of testable software in the application to determine whether they behave as expected.

Unit tests should not

- call out into (non-trivial) collaborators,
- access the network,
- hit a database,
- use the file system or
- spin up a thread.

Most of the unit tests can be found directory in the code documentation and are run using doctest. When they cannot be simple or extensible enough with impeding readability, they should be written in the `tests.unit` package.

Integration tests

Verify the communication paths and interactions between components to detect interface defects.

The line between unit and integration tests may become blurry. When in doubt, you are most certainly thinking integration tests. Write those in the `tests.integration` package.

Functional tests

Functional tests check a particular feature for correctness by comparing the results for a given input against the specification. They are often used as an executable definition of a user story. Write those in the `tests.functional` package.

Regression tests

A test that was written when a bug was found (and then fixed). It ensures that this specific bug will not occur again. The full name is *non-regression test*. It can also be a test made prior to changing an application to make sure the application provides the same outcome. Put these in the `tests.regression` package.

pymenu

pymenu package

Package main definition.

class `pymenu.`**`DictMenuEntry`**(*name*, *data*, *parent=None*)

Bases: `pymenu.MenuEntry`

A menu tree node made of a dictionary structure.

Parameters

- **`name`** (`str`) – The name of this node.
- **`data`** (`dict`) – The value of this node. If this is a dictionary, child nodes will be created from it.
- **`parent`** (`pymenu.MenuEntry`) – Parent entry node.

class `pymenu.`**`FileSystemMenuEntry`**(*path*, *parent=None*)

Bases: *`pymenu.MenuEntry`*

A menu tree node made from a filesystem path.

Parameters

- **`path`** (*`str`*) – Filesystem path from which to build the menu tree.
- **`parent`** (`pymenu.MenuEntry`) – Paren entry node.

Note: The creation of child nodes if **not lazy**. This means that creating an instance of this class from a top level folder of a large file sets will consumes a lot of RAM.

class `pymenu.`**`Menu`**(*root_entry*, *prompt*)

Bases: `object`

Parameters

- **`root_entry`** (`pymenu.MenuEntry`) –
- **`prompt`** (`pymenu.Prompt`) –

`choose_menu`()

Prompt for a choice of menu items.

Returns The chosen menu object

Return type *pymenu.Menu*

`choose_value`()

Prompt until a leaf menu item is choosen.

Returns The associated value for the chosen item.

Return type Any

`entry`

Returns – pymenu.MenuEntry

class `pymenu.`**`MenuEntry`**(*name*, *value=None*, *parent=None*)

Bases: `anytree.node.Node`

A menu tree node.

This is essentially a tree node, as inherited by `anytree.node.Node`. Sub classes may require to implement the creation of child entries.

Parameters

- **`name`** (*`str`*) – A name for this node.
- **`value`** (*`Any`*) – Associated value.
- **`parent`** (`pymenu.MenuEntry`) – Parent entry node.

`value`

class `pymenu.`**`Prompt`**

Bases: `object`

Abstract class for defining menu user interfaces.

`prompt_for_one`(*choices*)

Parameters **`choices`** (*`list`*) – List from which to choose from.

Returns str

class pymenu.**SimpleCommandPrompt**(*question=None*, *prompt=None*)

Bases: *pymenu.Prompt*

Simply prompt a user for choices in command line.

Parameters

- **question** (*str*) – Header question displayed before the choices.
- **prompt** (*str*) – Actual prompt message for the user.

prompt_for_one(*choices*)

Parameters **choices** (*list*) – List from which to choose from.

Returns Chosen key

Return type str

Raises KeyError – when the select item in not a valid choice.

Subpackages

pymenu.ext package

Subpackages

pymenu.ext.pyxdg package

class pymenu.ext.pyxdg.**Application**(*entry*, *parser=None*, *term_args=None*)

Bases: object

A launchable application defined by a XDG desktop entry.

Parameters

- **entry** (*xdg.Menu.MenuEntry*) – The desktop entry for this application.
- **parser** (*Callable*) – A function that parses an Exec string of a desktop entry and returns an abstract syntax tree (AST) of it. The AST is expected to be made of lists and have the following structure (given the input app arg1 arg2):

```
[
    ['a', 'p', 'p'],
    [
        ['a', 'r', 'g', '1'],
        ['a', 'r', 'g', '2']
    ]
]
```

  The default parser should work in most cases.

- **term_args** (*list*) – Command line argument prefixes for terminal applications. In XDG compliant desktop environments, the default (['x-terminal-emulator', '-e']) should be enough since it work on any setup that implements the Debian Alternatives System which is common in many UNIX distributions and most popular desktop environments.

  If you do not use this from of a XDG compliant environment (in Qtile, for instance) you will need to set this manually.

`arguments`

Provide the command line arguments for this application.

Some (`%i`, `%c`, `%k`) fieldcode placeholders are replaced. Target-like fieldcodes placeholders like `%f`, `%F`, `%u` and `%U` are not replaced.

Returns list

`entry`

`executable`

Provide the command line executables part for this application.

This may include terminal-specific executables and arguments, such as `['x-terminal-emulator', '-e']` in addition to the actual executable if this is a terminal application.

Returns list

`launch`(**target_uris*, ***popen_kwargs*)

Launch this application with provided targets.

Parameters

- **`*target_uris`** – Positional arguments are used as URI targets for this application. If this application can handle multiple URIs at once, they are all parametrized in one subprocess. If this application can only handle one URI at a time, multiple processes are launched. If this application cannot handle target URIs, this argument is ignored.
- **`**popen_kwargs`** – This application is launched as subprocesses using `subprocess.Popen`. These keyword arguments are simply passed along to this subprocess constructor.

Returns All subprocesses launched.

Return type list

class `pymenu.ext.pyxdg.`**`XdgMenuEntry`**(*wrapped_entry*, *app_factory=None*, *parent=None*)

Bases: *`pymenu.MenuEntry`*

Wrap an XDG menu entry.

Parameters

- **`wrapped_entry`** – An object defined in the `xdg.Menu` module.
- **`app_factory`** (*`Callable`*) – A function that takes a `xdg.Menu.MenuEntry` and returns a *`Application`*.
- **`parent`** –

See also:

`xdg.Menu.Menu`

classmethod **`from_xdg_menu_file`**(*menu_def_file*)

Constructor for a *.menu* file.

See also:

`make_xdg_menu_entry()`

`pymenu.ext.pyxdg.`**`exec_parser`**(*exec_string*)

Make the AST for a XDG Exec string.

Parameters **`exec_string`** (*`str`*) –

Returns AST

Return type list

pymenu.ext.pyxdg.**launch_xdg_menu_entry**(*entry*, **targets*)

A convenient launcher for desktop entries.

This uses the *Application* with default values.

Parameters **entry** (*xdg.Menu.MenuEntry*) –

Returns None

pymenu.ext.pyxdg.**make_xdg_menu_entry**(*menu_def_file=None*, *cls=None*)

Make a *pymenu.MenuEntry* based on a XDG .menu file.

This is usually located in `/etc/xdg/menus/applications.menu`.

Parameters

- **menu_def_file** (*str*) – Path to a *.menu* file as defined in the Desktop Menu Specification. Defaults to `/etc/xdg/menus/applications.menu`

  This file can usually be found in the `/etc/xdg/menus` folder. The following command is a good start to list these .menu files:

  These *.menu* file may not include applications that installed their desktop entries in a user folder such as `~/.local/share/applications`. In order to add additional directories to the desktop entries search path, you need to add a `<AppDir>` tag to the *.menu* file for the relevant directory.

- **cls** (*type*) – The subclass of *pymenu.MenuEntry* to create. The default is *XdgMenuEntry*.

See also:

pymenu.MenuEntry

pymenu.ext.xdmenu package

class pymenu.ext.xdmenu.**DmenuPrompt**(*dmenu=None*)

Bases: *pymenu.Prompt*

Parameters **dmenu** (*xdmenu.BaseMenu*) –

prompt_for_one(*menu*)

Parameters **menu** (*list*) – List from which to choose from.

Returns str

tests package

Subpackages

tests.functional package

tests.integration package

tests.regression package

tests.unit package

Subpackages

tests.unit.ext package

Submodules

tests.unit.ext.test_pyxdg module

class `tests.unit.ext.test_pyxdg.`**`TestData`**(*input*, *expected*)

Bases: `object`

`tests.unit.ext.test_pyxdg.`**`exec_string`**(*request*)

`tests.unit.ext.test_pyxdg.`**`test_exec_parser`**(*exec_string*)

Check that the exec_parser properly parse the input from exec_string.

Parameters **`exec_string`** (`TestData`) –

Returns:

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index