Pylease Documentation

Release 0.2

Bagrat Aznauryan

June 30, 2015

Contents

If you are working on a fast-changing Python project which needs frequent releases, you might get in trouble by repeating some routine tasks again and again. These tasks include updating the package version, creating tags in the source repository, uploading to PyPi or to your own private repository, etc.

To escape this manual mess, Pylease comes eager to help you out by making those processes as simple as possible by requiring as little as possible.

This documentation includes all the information needed for you to use Pylease, as well as extend it further.

CHAPTER 1

Introduction

Pylease works on Python projects that are being managed by setuptools, i.e. have `setup.py` file in their root directory. This is enough to make Pylease get things done. Pylease is an extensible modular tool, which enables the developers to enhance it further. So let us start with the installation first.

1.1 Installation

Pylease is a regular Python package which comes with a command line tool, obviously called `pylease`. So to start:

```
$ pip install pylease
```

And then just check if everything went fine:

```
$ pylease --version
Pylease version 0.2
```

1.2 Workflow

The simplicity of Pylease is that it does not require any specific configuration or scripts, it accepts Python projects just as-is. Although, you can configure Pylease to fully use its features, it is not required and you can just start using Pylease on a project you were working on for years. So let us consider two scenarios.

1.2.1 Project from Scratch

So to start a new Python project, just create a new empty directory for your project and then use the `init` command to create a skeleton for your project:

```
$ mkdir my_project
$ cd my_project
$ pylease init my_project
```

Now you have the skeleton of your project ready to be used:

```
$ ls -LR
my_project setup.cfg  setup.py

./my_project:
__init__.py
$ cat setup.py
```

```
import my_project
from setuptools import setup

setup(name='my_project',
      version=my_project.__version__)
$ cat setup.cfg
[pylease]
version-files = my_project/__init__.py
$ cat my_project/__init__.py
__version__ = '0.0'
```

1.2.2 Existing Project

If you have an already initialised Python project, then the first thing you will want to do for feeling the presence of Pylease, is the following:

```
$ cd /path/to/your/project/root
$ ls
... setup.py ...
$ pylease status
Project Name: <your project>
Current Version: <project version>
```

1.2.3 Releasing a Project

The idea of Pylease came while doing some routine tasks during a release process of a Python project, thus the main focus of it is the release process itself. So Pylease provides the `make` command to perform an appropriate release. But before doing anything release related, first check out your current status as it is done for an *Existing Project*.

So while you are working on a project, the current version defined is the last version the project was released with. As you can see in initialising a *Project from Scratch*, the initial version is `0.0`, i.e. no release is done yet.

You may ask a reasonable question, why does not the current version represent the version you want to release next? The reason for that is that it is possible that while working on the project, you might have a minor bug, small feature that will need a rapid release, so you will need to make a `patch` or `minor` level release.

As already mentioned, you perform a release with the help of the `make` command of Pylease. The main and required parameter of `make` command is the release level, which is passed as one of the following:

- `--major`
- `--minor`
- `--patch`
- `--dev`

For instance, executing `pylease make --minor` on a project with version `0.3` will update it to `0.4`:

```
$ pylease status
Project Name: example
Current Version: 0.3
$ pylease make --minor
$ pylease status
Project Name: example
Current Version: 0.4
```

So this is pretty much all what the release is. As a result, you will have your `setup.py` updated to the new version. To customize the behaviour of the release process, you might want to take a look at Pylease configuration.

Moreover, if you wish to add your own custom actions to Pylease, you should definitely get into extending Pylease.

For a quick reference, always consider to take a look at `--help` messages for commands, e.g. `pylease make --help`.

CHAPTER 2

Built-in Plugins

As you will see in the Extensing Pylease section, you can customize and extend Pylease the way you like. Fortunately, Pylease comes batteries included, with built-in plugins for most common tasks. Now let us take a look at each plugin separatelly.

2.1 Git

Pylease supports integration with git. You can enable this plugin with the `--git-tag` option of the `make` command. Consider the following situation:

```
$ pylease status
Project Name: example
Current Version: 0.3
$ git tag -l
v0.1
v0.1.1
v0.2
v0.3
```

As you can see the **example** project has four releases - `0.1`, `0.1.1`, `0.2`, `0.3`, and current version is `0.3`. So imagine you want to make a `patch` release and create appropriate tag in your git repository:

```
$ pylease make --patch --git-tag
$ pylease status
Project Name: example
Current Version: 0.3.1
$ git tag -l
v0.1
v0.1.1
v0.2
v0.3
v0.3.1
```

2.2 PyPI

The Pylease PyPI plugin enables to automatically upload your package egg to PyPI. The only prerequisite for this action is that the package name must be already registered, as well as you must be responsible for the authentication over the .pypirc file.

To enable this feature during the release process you should just use the `--pypi` option like this:

```
$ pylease make --major --pypi
```

This `--pypi` part of this command simply does the same as the `python setup.py sdist upload`.

CHAPTER 3

Pylease Configuration

As you have already seen, Pylease works on an existing project without requiring any configuration. However, at some point in time you will need to make some configuration to use Pylease fully. The place for Pylease configuration is the `setup.cfg` file of your project, under the `[pylease]` section. Here is an example configuration:

setup.cfg:

```
...

[pylease]
version-files = my_project/__init__.py

...
```

Following is the list of all configuration parameters with their descriptions.

`version-files` A list of files where the version must be updated to the new one. Here is an example value for this parameter:

```
version-files = my_project/__init__.py, setup.py
```

`use-plugins` A list of external plugins to load. If you have installed `example_plugin` package in your Python environment, and want Pylease to use that plugin, you just need to add the `example_plugin` name to the list of this parameter.

CHAPTER 4

Extending Pylease

Although Pylease is an Open Source project, it is impossible to consider and include all possible features at once. Moreover, one may want a particular feature urgently. Even after sending a feature request, he may have no time to wait for confirmation, implementation and release of that feature. On the other hand, Pylease may get too much loaded with features that does not get used by everyone.

For this reason, one of the major parts of Pylease is its modular extension system. It allows to develop a separate Python pakage and optionally plug it into Pylease.

So basically there are two ways to extend Pylease:

- Add before and after tasks for existing commands
- Add new commands

The following two subsections discuss both scenarios with their details.

4.1 Extending Existing Commands

Here is the step-by-setp guide on how to add extensions to Pylease.

The first step is inheriting the *Extension* class and adding it to your package `__init__.py`, where you will place the initialisation code by implementing the *load()* method. As an instance attribute you will have the *_lizy* attribute, which is an instance of *Pylease* class, and contains everything you need for your extension.

Extending an existing Pylease command is done by adding a *BeforeTask* or *AfterTask* (or both) to it using the *Command* methods *add_before_task()* and *add_after_task()*. What is needed to do is just implement those classes and add their instances to the command that is the subject of extension. For both *BeforeTask* and *AfterTask* you need to inherit and implement their *execute()* method, which must include the extension logic. Also, in case of *AfterTask*, you are provided with the *_command_result* attribute, which is the result returned by the command being extended.

So basically this is the scenario of extending a Pylease command:

- Inherit and implement *BeforeTask* or/and *AfterTask*
- Inherit *Extension*
- In the *load()* implementation get the corresponding *Command* instance from the *_lizy* singleton
- Add the *BeforeTask* or/and *AfterTask* instances to the command instance

4.2 Adding New Commands

To add a new command to Pylease it is enough to implement a class by inheriting the *Command* class and add it to your package `__init__.py`. Implementing the *Command* class is implementing the *_process_command()* method. As an additional convenience you can inherit the *NamedCommand* class instead. This will eliminate the need to manually specify the name of the command while calling the base constructor. Instead this base class will automatically parse the command name from the class name by removing the "Command" suffix and using the rest as the command name. So for example, the `init` command is defined as a child class of *NamedCommand* with the name `InitCommand`.

As in the case of implementing *Extension*, here you will also be provided with the *Pylease* `lizy` singleton.

4.3 Rollbacks

Even if you ship a perfectly clear extension, which will never crash in any conditions, you have no guarantee for others. As Pylease is a modular tool, it is possible to plug any number of independent extensions. This means that it is possible that after your extension task or command is executed, there may be another task executed after, that will lead to an error. In this case, you might need to rollback all the changes that your extension made to maintain the original state of the project.

For instance, the Git plugin makes a commit for the changes of version of the project, then creates a tag for the version. If any error raises after this operations, is it critical to roll them back. Thus, this plugin deletes the last commit and removes the created tag.

Pylease provides the *Rollback* class and *Stage* decorator to implement this feature in your extension. The *Stage* decorator enables to have a staged rollback. For example, in case of the Git plugin, if the error occurres in the stage of creating the version tag, the only rollback step to perform is deleting the last commit.

For reference on using this classes please see the Class Reference for *Rollback* and *Stage*

CHAPTER 5

Class Reference

class pylease.**Pylease**(*parser*, *cmd_subparsers*, *info_container*)

The main class of Pylease, which contains all the needed resources for extensions. This class is initialised once and by Pylease, which is the so called `lizy` object. It is passed to *Command* and *Extension* instances.

info_container
pylease.InfoContainer

Contains information about current status of the project. Minimal information is `name` and `version`.

commands
dict

A dictionary of Pylease commands, including commands defined in extensions if any. The values of the dictionary are instances of *Command* class.

parser
argparse.ArgumentParser

The root parser of Pylease. Use this object to add command line arguments to Pylease on the same level as `--version` and `--help`.

config
dict

A dictionary representing the configuration parsed from `setup.cfg` defined under `[pylease]` section. If a configuration value in the configuration file is defined as `key1 = valA, valB, valC` then the value of the `key1` key of this attribute will be an instance of `list` and be equal to `['valA', 'valB, 'valC']`.

class pylease.**InfoContainer**

A simple container that maps a provided dictionary to its attributes. This provides the current status of the project, and the minimal built-in information attributes are the following:

name
str

The name of the project.

version
str

The current versin of the project

is_empty
bool

The status of current working directory, i.e. indicates whether it is empty or not.

set_info(***kwargs*)
Used to extend the information about the project.

Example

Below are two options on how to use/extend the `InfoContainer`:

```
info = InfoContainer()

# Option 1
info.set_info(info1='value2', info2='value2')

# Option 2
more_info = {'info3': 'value3'}
info.set_info(**more_info)

# Then you can access your info as instance attributes
print(info.info2)  # will print 'value2'
```

class pylease.ext.**Extension**(*lizy*)
The entry point to implementing Pylease extensions. Pylease loads subclasses of this class and invokes the *load()* method.

_lizy
pylease.Pylease

The *Pylease* singleton, that is initialised and passed to all subclass instances.

load()
This method is being called by Pylease when all the extensions are being loaded. All the initialisation code must be implemented in the body of this method.

class pylease.cmd.task.**BeforeTask**(*rollback=None*)

execute(*lizy*, *args*)
The place where the extension logic goes on.

Parameters

- **lizy** (pylease.Pylease) – The *Pylease* singleton that provides all the needed information about the project.
- **args** (*argparse.Namespace*) – The arguments supplied to the command line.

enable_rollback(*rollback=None*)
Enables a rollback for the task. The provided rollback gets executed in case of failure in the execution stack.

rollback
pylease.cmd.rollback.Rollback

The rollback instance for the task.

class pylease.cmd.task.**AfterTask**(*rollback=None*)

execute(*lizy*, *args*)
The place where the extension logic goes on.

Parameters

- **lizy** (pylease.Pylease) – The *Pylease* singleton that provides all the needed information about the project.
- **args** (*argparse.Namespace*) – The arguments supplied to the command line.

_command_result
A dictionary containing information by the completion of the command execution.

class pylease.cmd.**Command**(*lizy*, *name*, *description*, *rollback=None*, *requires_project=True*)
This class is one of the main point of Pylease. For adding new commands just inherit from this class and implement *_process_command()* method.

__init__(*lizy*, *name*, *description*, *rollback=None*, *requires_project=True*)
This constructor should be called from child classes and at least be supplied with at least `name` and `description`.

Parameters

- **lizy** (pylease.Pylease) – The *lizy* object, which is initialized and passed by Pylease.
- **name** (*str*) – The name of the command, which will appear in the *usage* output.
- **description** (*str*) – Description of the command which will also appear in the help message.
- **rollback** (pylease.cmd.rollback.Rollback) – The rollback object that will be executed in case of failure during or after the command. This parameter may be emitted if the command does not need a rollback, or may be set in the process of command execution using the *enable_rollback()* method, if it depends on some parameters during runtime.
- **requires_project** (*bool*) – Boolean indicating whether the command requires to operate on an existing project. E.g. the `init` command requires an empty directory.

_process_command(*lizy*, *args*)
The method which should be implemented when inheriting the *Command*. All the command logic must go into this method.

Parameters

- **lizy** (pylease.Pylease) – The *Pylease* singleton.
- **args** (*argparse.Namespace*) – The arguments passed to the command line while invoking Pylease.

add_before_task(*task*)
Adds a *BeforeTask* to the *Command*.

Parameters **task** (pylease.cmd.task.BeforeTask) – The task to be added.

add_after_task(*task*)
Adds a *AfterTask* to the *Command*.

Parameters **task** (pylease.cmd.task.AfterTask) – The task to be added.

class pylease.cmd.**NamedCommand**(*lizy*, *description*, *rollback=None*, *requires_project=True*)
Same as the *Command* class, however this class enables a little taste of convenience. You can define a class having name with a suffix "Command" and it will automatically assign the prefix of the class name as the command name.

__init__(*lizy*, *description*, *rollback=None*, *requires_project=True*)
Same as *__init__()* of *Command* class, except that the `name` argument is passed automatically.

class pylease.cmd.rollback.**Rollback**
This class provides a facility to define a staged rollback process. The scenario of using this class is the following:

1. Inherit *Rollback* class
2. Define rollback stages as instance methods
3. Decorate each rollback method with *Stage* decorator, by specifying stage name and priority
4. Enable each stage separately calling *enable_stage()* method

enable_stage(*stage*)
Enable particular stage by name.

Parameters **stage** (*str*) – Stage name to enable.

rollback()
Execute all rollback stages ordered by priority.

class pylease.cmd.rollback.**Stage**(*stage*, *priority=0*)
Decorator used in custom *Rollback* classes for associating each method with a stage, and setting priority.

Parameters

- **stage** (*str*) – The name of the stage.
- **priority** (*int*) – The order priority of the stage to be rolled back. Defaults to 0.

Example

Here is an example of how to use the *Stage* decorator in combination with the *Rollback* base class:

```
class ExampleRollback(Rollback):
    @Stage('some_stage', 1)
    def some_stage_with_priority_1(self):
        pass  # your some_stage rollback goes here
```

p

Symbols

A

B

C

E

I

L

N

P

R

S

V