pydeploy Documentation

Release 0.2.2

Rotem Yaari

September 08, 2011

CONTENTS

Contents:

CHAPTER

ONE

PYDEPLOY OVERVIEW

1.1 The Problem

Python is a great language and development platform, constantly gaining enormous popularity in various fields of the industry.

Any software development platform (and languages/interpreters in particular) need to provide their developers with means to reuse code that was written elsewhere. In most cases this is done in the form of *modules* (or *packages*). It is generally agreed that software components should be decomposed and granularized so that each package is reusable in as many contexts as possible, while still being coherent and maintainable.

However, an inherent problem stems from this - how do you keep track of all the packages and modules that your code needs? If you want to use a specific library **X**, how do you make sure you install it properly, so that all of its dependencies are intact?

This issue of distribution and installation is very important, and each platform addresses it individually. When focusing on Python specifically, we can see that there's far from one golden solution. A quick count shows:

- Distutils – Python's original, out-of-the-box packaging and distribution solution
- setuptools – Enhances distutils, and adds the *easy_install* tool for automatic installation from pypi
- distribute – a fork of setuptools (and a drop-in replacement). It is related to the *pip* tool, which is a replacement for some of the uses of *easy_install*.

These tools work great (in general) if you're distributing something that will be available on the internet through the PyPI indexing site. Your **setup.py** file can include requirements identified by project names and versions, and *easy_install* / *pip* take care of installing all dependencies from PyPI.

However, erious hassle forms when you try to deploy packages that are written in-house, without being published. If you have plenty of packages, some arriving from the outside world and some arriving from within your organization, you're stuck, because the same mechanism cannot work here (more on that in a second).

Even if you do solve the installation issue, there is another issue of *deployment*. For example, in some cases it might not be enough to just run **easy_install my_package**:

- You might not have permissions to run easy_install (in many cases the global easy_install requires root permissions), in which case you might need a *virtual environment <http://pypi.python.org/pypi/virtualenv>*.
- You might want to run some more tools or scripts on the environment after the installation (for instance, run automated tests, servers or what not)
- You might want to do different actions depending on the platform on which you're installing.

All the tools mentioned above are not very easily automatable, so assuming you want to use Python and not some variant of shell scripting, the deployment tasks become inconvenient.

1.2 Possible Solutions

1.2.1 A Custom PyPI Server

This approach generally works in many places, but is very error-prone. It is mostly intended to solve the installation issue, rather than the deployment issue.

In this scheme you establish a server inside your organization that mimicks **PyPI**. You then register your internal packages to that server, and pass a flag to *easy_install* to install from that repository.

However, although installing from such a server is very simple (and not error prone, because at the worst case you simply won't find your package), distributing internal packages through this mechanism is simply unacceptable as a solution. In order to register and/or upload your package you need a .pypirc file. If you get any field of that file wrong (which is quite easy, I assure you), you'll silently be defaulted to the global pypi server.

Assuming you distribute packages to both the internet and the internal organization, you'll soon end up accidentally publishing internal packages. Yikes.

Also, if you cache packages in the local mimicking server, you start dealing with inconsistencies once the global packages get updated, which is also not desirable.

1.2.2 pip

Pip technically has the capability of installing directly from source code repositories and from URLs. However, eventually it still executes the same logic as *easy_install*, meaning that it will be difficult and obscure to declare dependencies from internal projects in your setup.py file.

Pip also doesn't take care of the deployment part.

1.2.3 zc.buildout

zc.buildout <http://pypi.python.org/pypi/zc.buildout> is a complex project which acts as a mix between a build system, a deployment system and a virtual environment. It basically solves most of the problems mentioned above by defining an environment with a configuration file pointing to actions to be taken, and "recipes", dictating what to do for each action.

This would have been a magic bullet for the problems listed above - however, it is still far from perfect.

First, the learning curve is far from optimal. If you look for the documentation for zc.buildout, you'll stumble upon multiple sources, like this. However, the best source people are pointing to is this one (written for an older version), and from the length of it, you can see something is not right. The document uses mostly doctests to describe what's happening, but while doctests are great, they do a poor job at describing how to write a buildout configuration file.

Second, zc.buildout is extremely slow. If you have an environment you would like to constantly update, each build takes a lot of time, and can cross the 1 minute threshold in some cases.

Third, zc.buildout is extremely complex, and just like any complex project it has a lot of bugs that are not getting solved in time. You can see many people already working on forked versions of zc.buildout as their open issues don't get resolved. With all due respect to zc.buildout, the task we would like to perform is simpler, and makes it an overkill.

1.3 Pydeploy

Pydeploy tries to bridge the gap between installation and deployment, and at the same time provide a reasonable (although not perfect yet) mechanism for mixed in-house and public package management:

- Pydeploy "configuration" is actually a python script, so you are not confined to the syntax of a .ini file
- The simplest use case is the simple: **pydeploy install <package>**. No learning curve if you just want to install a single package from somewhere.

CHAPTER

TWO

FOR USERS (QUICK GUIDE)

2.1 Installing a Package

Installing a package with pydeploy is as easy as it gets:

```
pydeploy install <package>
```

package can be either one of:

- a local package directory (containing a **setup.py** file)
- a URL of a package (compressed) to download and install
- a URL of an SCM repository (like git://some_hostname/repo)
- a name of a package to be searched in PyPI (in this case it's actually invokes *easy_install* under the hood).

Note: if you would like the package to be installed in a virtual environment (with **virtualenv**), you can use the **–env=path** flag to point it to your virtual environment directory.

2.2 Running a Configuration File

In some cases pydeploy is used to run a file which dictates what steps are to be done. In this case you should just do:

```
pydeploy run <file>
```

file can be either:

- a local file on your filesystem
- a URL to a file on a server somewhere

Note: the –env flag exists for the **run** command as well. It will execute the file in the context of the virtual environment specified. See *the next chapter* for more details.

2.3 Bootstrapping Pydeploy

On hosts that don't have pydeploy installed (and perhaps installation is an issue due to permissions), a bootstrapper for pydeploy is available online:

```
wget https://github.com/vmalloc/pydeploy/raw/master/scripts/bootstrapper.py
python bootstrapper.py <args>
```

Where *args* are the argument you would pass to pydeploy. You can even do this in one line:

```
curl https://github.com/vmalloc/pydeploy/raw/master/scripts/bootstrapper.py | python - <args>
```

Or even shorter:

```
curl -L bit.ly/ilTVUN | python - <args>
```

CHAPTER

THREE

FOR DEVELOPERS / PACKAGE MAINTAINERS

3.1 Writing Pydeploy Files

A “pydeploy file” is simply a Python script that can be executed by pydeploy. If you write a file – **test.py** – containing the following:

```
print "hello"
```

And execute it using:

```
pydeploy run test.py
```

You’ll simply see the “hello” message printed out. What distinguishes pydeploy scripts from normal Python files is that they get a special variable in their available namespace – the **env** object, pointing to the `PythonEnvironment` class. The environment represents the current Python installation which we’re working on. The basic entry point to it is `PythonEnvironment.install()`. This method receives a *source* to install, which can be many things:

```
env.install(URL("http://path/to/dependency.tgz"))
env.install(SCM("git://some/url/to/package"))
env.install("/path/to/package")
```

Accepted sources are:

- Path (default): local path to the package
- EasyInstall (default if argument is not a path and not a URL): for installing using easy_install
- PIP: install using pip
- SCM: for installing from source repository
- URL: for installing from the web

Now once you run your file, you’ll trigger the installation of these packages in your environment (whether global or a virtual environment).

3.2 Using Pydeploy Files when Installing Packages

Pydeploy is mostly useful when used to install packages. In this scenario, you do not execute a pydeploy file directly, but rather use pydelpoy’s *install* command to install a package.

Whenever a package contains a file called *pydeploy_setup.py* (located next to its *setup.py* file), it will be run by pydeploy prior to actually installing the package.

This rule applies recursively. Let's say that we have a package *A*, that depends on package *B*, stored locally in a git repository. Furthermore let's say *B* depends on *C* in a different repository. You can write the following *pydeploy_setup.py* in *A*:

```
env.install("git://path.to.server/for/B.git")
```

And in turn, write the following *pydeploy_setup.py* in *B*:

```
env.install("git://another.path.to/C.git")
```

When you install package *A* (regardless of where from), pydeploy will:

```
1. run the pydeploy_setup for A
2. follow the install() call to download and install B
3. prior to B's installation, it will execute B's pydeploy_setup.py
4. C will be downloaded and installed
5. B will be installed
6. A will be installed
```

3.3 Installing Dependencies Through *install_requires*

When installing from sources other than *easy_install* or *pip*, pydeploy can extract the names of the packages which are dependencies of the package being installed. By default this information is not used, but you can *alias* a package name to a specific source, making the *install_requires* feature work for you.

Let's take a simple case of a package *A* depending on *B* and *C*. Let's also assume that *B* and *C* each in turn depend on *D*. We don't want to repeat *D*'s URL in both setup files of B and C, but we still have to make the installation work. How do we make that happen?

We'll create a common file in a central location. This central location can be an SCM server, a network share, or a config file in the user's homedir, whichever best suits you. Let's call it *pydeploy_aliases.py*:

```
env.add_alias("A", "/path/to/a/")
env.add_alias("B", "http://bla.com/B.tgz")
env.add_alias("C", "git://git.server.for/C.git")
env.add_alias("D", "http://download.server.for/D.tgz")
```

The file uses the `add_alias()` method to let pydeploy know that the package name 'X' refers actually to a custom source, rather than to PyPI's index.

And now all we need to do is point our *pydeploy_setup.py* files to that alias file. We use `execute_deployment_file_once()` for that:

```
# in each pydeploy_setup.py:
env.execute_deployment_file_once("http://download.server.for/pydeploy_aliases.py")
```

Now when installing A, pydeploy will analyze its dependencies, detect B and C (which it now knows are aliases) and install them. In the process it will also detect D and install it according to its alias.

3.4 Misc. Utilities

3.4.1 SSH Remote Deployment

The *pydeploy.remote* utility module provides a manner for deploying a script remotely.

```
from pydeploy.remote import deploy_via_ssh
return_code = deploy_via_ssh("hostname", "http://pydeploy_file_url", "/tmp/deployment_dir")
```

The *deploy_via_ssh* utility can also receive file objects with the script to run, as a convenience:

```
from pydeploy.remote import deploy_via_ssh
from cStringIO import StringIO
return_code = deploy_via_ssh("hostname", StringIO("print 'source here!'"), "/tmp/deployment_dir")
```

CHAPTER

FOUR

API REFERENCE

4.1 The Environment Class

class `pydeploy.environment.`**`PythonEnvironment`**(*argv=()*)

`checkout`(*source*, **args*, ***kwargs*)
Downloads a package from *source*, and returns the path to which it was extracted.

`install`(*source*, *reinstall=True*)
Installs a package from *source*.

@param reinstall: If False, skips installation if this source has already been installed.

CHAPTER

FIVE

FAQ

placeholder

CHAPTER

SIX

KNOWN ISSUES

- When using PIP to install a library that exists on the host, pip will not perform an actual installation. This means, for instance, that scripts will not be copied to the bin dir of the virtual environment. In such cases EasyInstall is recommended.

CHAPTER

SEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX