DebTools Documentation

Release 0.7.3

Matthieu Gallet

October 30, 2016

Contents

Overview:

CHAPTER 1

Full table of contents

1.1 Installing

DebTools is compatible with Python 2.7.x and Python 3.2+. Two dependencies are required:

- stdeb
- backports.lzma (only required on Python 2.7 and 3.2/3.3)

The easiest way is to use pip:

```
pip install debtools
```

If debtools was already installed:

```
pip install debtools --upgrade
```

If you prefer install directly from the source:

```
cd DebTools
python setup.py install
```

1.2 deb-dep-tree

Download packages and show the dependencies of a given package:

```
deb-dep-tree libgcc1_4.7.2-5_amd64.deb
libgcc1
=======

  * multiarch-support
  * gcc-4.7-base (= 4.7.2-5)
  * libc6 (>= 2.2.5)
```

Ok, nothing new from the standard *dpkg -I libgcc1_4.7.2-5_amd64.deb* command, but you can provide either a package name or a .deb filename:

```
deb-dep-tree libgcc1
Réception de : 1 Téléchargement de libgcc1 1:4.7.2-5 [43,1 kB]
43,1 ko réceptionnés en 0s (45,2 ko/s)
libgcc1
=======
```

```
  * multiarch-support
  * gcc-4.7-base (= 4.7.2-5)
  * libc6 (>= 2.2.5)
```

The package will be downloaded in the current directory. You can recursively retrieve all dependencies.

```
deb-dep-tree libgcc1 -r
libgcc1
=======

  * multiarch-support
  * gcc-4.7-base (= 4.7.2-5)
  * libc6 (>= 2.2.5)

multiarch-support
=================

  * libc6 (>= 2.3.6-2)

libc-bin
========


gcc-4.7-base
============


libc6
=====

  * libc-bin (= 2.13-38+deb7u8)
  * libgcc1

$ ls
gcc-4.7-base_4.7.2-5_amd64.deb  libc6_2.13-38+deb7u8_amd64.deb  libc-bin_2.13-38+deb7u8_amd64.deb  li
```

Sometimes, there is a choice between several possibilities for a given dependency. These dependencies are ignored (since we cannot select one). However, you can use the *-l* flag to select choices which are currently installed on the system.

```
dpkg -I libssl1.0.0_1.0.1e-2+deb7u17_amd64.deb | grep Depends
Pre-Depends: multiarch-support
Depends: libc6 (>= 2.7), zlib1g (>= 1:1.1.4), debconf (>= 0.5) | debconf-2.0

dpkg -l | grep debconf
ii  debconf                          1.5.49                      all          Debian configuratio
ii  debconf-i18n                     1.5.49                      all          full internationali
ii  po-debconf                       1.0.16+nmu2                 all          tool for managing t

deb-dep-tree libssl1.0.0
libssl1.0.0
===========

  * multiarch-support
  * zlib1g (>= 1:1.1.4)
  * libc6 (>= 2.7)

deb-dep-tree libssl1.0.0 -l
```

```
libssl1.0.0
===========

  * debconf
  * multiarch-support
  * zlib1g (>= 1:1.1.4)
  * libc6 (>= 2.7)
```

You can also ignore some dependencies, by providing a file with a list of dependencies to ignore. Its format is the same as the output of the *dpkg -l* command.

```
dpkg -l | grep libc > /tmp/toignore
deb-dep-tree libgcc1 -r -i /tmp/toignore
libgcc1
=======

  * multiarch-support
  * gcc-4.7-base (= 4.7.2-5)
  * libc6 (>= 2.2.5)

multiarch-support
=================

  * libc6 (>= 2.3.6-2)

gcc-4.7-base
============
```

1.3 multideb

Create several Debian packages at once. Fetch the list of installed Python packages in the current virtualenv and package them as .deb packages using the standard *stdeb* tool. You can also:

- define the packages to create in a configuration file,
- specify options for any of these packages,
- run Python commands after archive expansion and between the creation of Debian source and the creation of the Debian package.

To create Debian packages for all currently installed Python packages, use the following command:

```
multideb --freeze
```

All options must be defined in a *stdeb.cfg* configuration file. In the [multideb-packages] section of *stdeb.cfg*, you can define extra packages to create: option name is the name of the package, option value is the required version. In the [multideb] section of *stdeb.cfg*, you can exclude some packages from .deb creation:

```
[multideb]
exclude = celery
    django
    gunicorn
```

You can define specific options for a given package. In addition of standard *stdeb* options, you can also define *pre_source* and *post_source* options. Values must be an importable Python function, which will be called with the following arguments *my_callable(package_name, package_version, deb_src_dir)*.

Here is the list of actions:

- download .tar.gz of the source code,
- expand this file,
- remove all *.pyc* files,
- run the *pre_source* function (if defined),
- run *python setup.py sdist_dsc*,
- run the *post_source* function (if defined),
- create the package with *dpkg-buildpackage*.

1.3.1 Usage

```
multideb
```

1.3.2 Callable hooks

pre_source hook is called just after the expand of the archive and the removal of compiled Python files (*.pyc*). The current working dir is changed to this directory (for example, *./setup.py* should exist) and *deb_src_dir* is *None* when this hook is called.

post_source hook is called after the *sdist_dsc* command. The current working dir is changed to archive directory (for example, *./setup.py* should exist) and *deb_src_dir* is valid when this hook is called. It corresponds to the single sub-directory in the directory *deb_dist*.

1.3.3 Sample config file

Here is a sample *stdeb.cfg* file:

```
[multideb-packages]
django = 1.8.3

[multideb]
exclude = funcsigs
    django-allauth
    gunicorn

[django]
pre_source = multideb.remove_tests_dir

[celery]
post_source = multideb.fix_celery

; list of standard stdeb options
[other_package]
Source = debian/control Source: (Default: <source-debianized-setup-name>)
Package = debian/control Package: (Default: python-<debianized-setup-name>)
Suite = suite (e.g. stable, lucid) in changelog (Default: unstable)
Maintainer = debian/control Maintainer: (Default: <setup-maintainer-or-author>)
Section = debian/control Section: (Default: python)
Epoc = version epoch
Depends = debian/control Depends:
Depends3 = debian/control Depends: for python3
Suggests = debian/control Suggests:
```

```
Suggests3 = debian/control Suggests: for python3
Recommends = debian/control Recommends:
Recommends3 = debian/control Recommends: for python3
Conflicts = debian/control Conflicts:
Uploaders = uploaders
Conflicts3 = debian/control Conflicts: for python3
Provides = debian/control Provides:
Provides3 = debian/control Provides: for python3
Replaces = debian/control Replaces:
Replaces3 = debian/control Replaces: for python3
Copyright-File = copyright file
Build-Conflicts = debian/control Build-Conflicts:
MIME-File = MIME file
Udev-Rules = file with rules to install to udev
Debian-Version = debian version (Default: 1)
Build-Depends = debian/control Build-Depends:
Forced-Upstream-Version = forced upstream version
Upstream-Version-Suffix = upstream version suffix
Stdeb-Patch-File = file containing patches for stdeb to apply
XS-Python-Version = debian/control XS-Python-Version:
Dpkg-Shlibdeps-Params = parameters passed to dpkg-shlibdeps
Stdeb-Patch-Level = patch level provided to patch command
Upstream-Version-Prefix = upstream version prefix
X-Python3-Version = debian/control X-Python3-Version:
MIME-Desktop-Files = MIME desktop files
Shared-MIME-File = shared MIME file
Setup-Env-Vars = environment variables passed to setup.py
```

1.4 aptenv

When your application is meant to be deployed using the official Ubuntu or Debian packages (like *python-django*). *aptenv* takes a list of Python packages (a standard requirements files, like the one produced by the *pip freeze command*) or the list of currently installed packages and fetch the list of available versions in the Ubuntu or Debian mirrors.

```
aptenv -u xenial -u xenial-updates --python 3 -r requirements.txt
```

You can also fetch the available Python version .. code-block:: bash

> aptenv -u xenial -u xenial-updates –python 3 -P python3.5 aptenv -u trusty -u trusty-updates –python 3 -P python3.4 aptenv -u precise -u precise-updates –python 3 -P python3.2 aptenv -u trusty -u trusty-updates –python 2 -P python2.7

By default, the debianized name of a Python package starts by *python-* or *python3-*. Some packages have a specific name. For example, the debian name of *ansible* is *ansible*. You can specify a file with all your exceptions, and the mapping for a few well-known Python packages is provided, you can use it with *-M*. You can also use this system for excluding some packages:

```
echo "PyYAML==3.12" > requirements.txt
aptenv -u xenial -u xenial-updates --python 3 -r requirements.txt
Unable to find any version for PyYAML
echo "PyYAML=python-yaml" > map
aptenv -u xenial -u xenial-updates --python 3 -r requirements.txt -m map
PyYAML==3.11
aptenv -u xenial -u xenial-updates --python 3 -r requirements.txt -M
PyYAML==3.11
```

```
echo "PyYAML=" > map
aptenv -u xenial -u xenial-updates --python 3 -r requirements.txt -m map
```

Then you can create a virtualenv corresponding to a plain Ubuntu Xenial or a Debian Stable installation:

```
aptenv -u xenial -u xenial-updates --python 3 -r requirements.txt > requirements-ubuntu-xenial.rst
mkvirtualenv ubuntu-xenial -p `aptenv -u xenial -u xenial-updates --python 3 -P` -r requirements-ubun
```

```
aptenv -u jessie -u jessie-backports --python 3 -r requirements.txt > requirements-debian-jessie.rst
mkvirtualenv ubuntu-xenial -p `aptenv -u jessie -u jessie-backports --python 3 -P` -r requirements-de
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search