deb-pkg-tools

Release 8.4

Mar 09, 2021

Contents

Welcome to the documentation of *deb-pkg-tools* version 8.4! The following sections are available:

CHAPTER 1

User documentation

The readme is the best place to start reading, it's targeted at all users and documents the command line interface:

1.1 deb-pkg-tools: Debian packaging tools

The Python package *deb-pkg-tools* is a collection of functions to build and inspect Debian binary packages and repositories of binary packages. Its primary use case is to automate builds.

Some of the functionality is exposed in the command line interface (documented below) because it's very convenient to use in shell scripts, while other functionality is meant to be used as a Python API. The package is currently tested on cPython 2.7, 3.5+ and PyPy (2.7).

Please note that *deb-pkg-tools* is quite opinionated about how Debian binary packages should be built and it enforces some of these opinions on its users. Most of this can be avoided with optional function arguments and/or environment variables. If you find something that doesn't work to your liking and you can't work around it, feel free to ask for an additional configuration option; I try to keep an open mind about the possible use cases of my projects.

Contents

1.1.1 Status

On the one hand the *deb-pkg-tools* package is based on my experiences with Debian packages and repositories over the past couple of years, on the other hand *deb-pkg-tools* itself is quite young. Then again most functionality is covered by automated tests; at the time of writing coverage is around 90% (some of the error handling is quite tricky to test if we also want to test the non-error case, which is of course the main focus :-)

1.1.2 Installation

The *deb-pkg-tools* package is available on PyPI which means installation should be as simple as:

```
$ pip install deb-pkg-tools
```

There's actually a multitude of ways to install Python packages (e.g. the per user site-packages directory, virtual environments or just installing system wide) and I have no intention of getting into that discussion here, so if this intimidates you then read up on your options before returning to these instructions ;-).

When *deb-pkg-tools* is being used to scan thousands of `*.deb` archives a significant speedup may be achieved using memcached:

```
$ pip install "deb-pkg-tools[memcached]"
```

Under the hood *deb-pkg-tools* uses several programs provided by Debian, the details are available in the *dependencies* section. To install these programs:

```
$ sudo apt-get install dpkg-dev fakeroot lintian
```

1.1.3 Usage

There are two ways to use the *deb-pkg-tools* package: As a command line program and as a Python API. For details about the Python API please refer to the API documentation available on Read the Docs. The command line interface is described below.

Usage: *deb-pkg-tools [OPTIONS] ...*

Wrapper for the deb-pkg-tools Python project that implements various tools to inspect, build and manipulate Debian binary package archives and related entities like trivial repositories.

Supported options:

Option	Description
`-i`, `--inspect=FILE`	Inspect the metadata in the Debian binary package archive given by `FILE` (similar to "dpkg `--info`").
`-c`, `--collect=DIR`	Copy the package archive(s) given as positional arguments (and all package archives required by the given package archives) into the directory given by `DIR`.
`-C`, `--check=FILE`	Perform static analysis on a package archive and its dependencies in order to recognize common errors as soon as possible.
`-p`, `--patch=FILE`	Patch fields into the existing control file given by `FILE`. To be used together with the `-s`, `--set` option.
`-s`, `--set=LINE`	A line to patch into the control file (syntax: "Name: Value"). To be used together with the `-p`, `--patch` option.
`-b`, `--build=DIR`	Build a Debian binary package with "dpkg-deb `--build`" (and lots of intermediate Python magic, refer to the API documentation of the project for full details) based on the binary package template in the directory given by `DIR`. The resulting archive is located in the system wide temporary directory (usually /tmp).
`-u`, `--update-repo=DIR`	Create or update the trivial Debian binary package repository in the directory given by `DIR`.
`-a`, `--activate-repo=DIR`	Enable "apt-get" to install packages from the trivial repository (requires root/sudo privilege) in the directory given by `DIR`. Alternatively you can use the `-w`, `--with-repo` option.
`-d`, `--deactivate-repo=DIR`	Cleans up after `--activate-repo` (requires root/sudo privilege). Alternatively you can use the `-w`, `--with-repo` option.
`-w`, `--with-repo=DIR`	Create or update a trivial package repository, activate the repository, run the positional arguments as an external command (usually "apt-get install") and finally deactivate the repository.
`--gc`, `--garbage-collect`	Force removal of stale entries from the persistent (on disk) package metadata cache. Garbage collection is performed automatically by the deb-pkg-tools command line interface when the last garbage collection cycle was more than 24 hours ago, so you only need to do it manually when you want to control when it happens (for example by a daily cron job scheduled during idle hours :-).
`-y`, `--yes`	Assume the answer to interactive questions is yes.
`-v`, `--verbose`	Make more noise! (useful during debugging)
`-h`, `--help`	Show this message and exit.

One thing to note is that the operation of `deb-pkg-tools --update-repo` can be influenced by a configuration file. For details about this, please refer to the documentation on deb_pkg_tools.repo.select_gpg_key().

1.1.4 Dependencies

The following external programs are required by *deb-pkg-tools* (depending on which functionality you want to use of course):

Program	Package
apt-ftparchive	apt-utils
apt-get	apt
cp	coreutils
dpkg-deb	dpkg
dpkg-architecture	dpkg-dev
du	coreutils
fakeroot	fakeroot
gpg	gnupg
gzip	gzip
lintian	lintian

The majority of these programs/packages will already be installed on most Debian based systems so you should only need the following to get started:

```
$ sudo apt-get install dpkg-dev fakeroot lintian
```

1.1.5 Platform compatibility

Several things can be tweaked via environment variables if they don't work for your system or platform. For example on Mac OS X the `cp` command doesn't have an `-l` parameter and the `root` user and group may not exist, but despite these things it can still be useful to test package builds on Mac OS X. The following environment variables can be used to adjust such factors:

Environment variable	Default value
$DPT_ALLOW_FAKEROOT_OR_SUDO	true
$DPT_CHOWN_FILES	true
$DPT_FORCE_ENTROPY	false
$DPT_HARD_LINKS	true
$DPT_PARSE_STRICT	true
$DPT_RESET_SETGID	true
$DPT_ROOT_GROUP	root
$DPT_ROOT_USER	root
$DPT_SUDO	true

Environment variables for boolean options support the strings `yes`, `true`, `1`, `no`, `false` and `0` (case is ignored).

Disabling sudo usage

To disable any use of `sudo` you can use the following:

```
export DPT_ALLOW_FAKEROOT_OR_SUDO=false
export DPT_CHOWN_FILES=false
export DPT_RESET_SETGID=false
export DPT_SUDO=false
```

1.1.6 Contact

The latest version of *deb-pkg-tools* is available on PyPI and GitHub. The documentation is hosted on Read the Docs. For bug reports please create an issue on GitHub. If you have questions, suggestions, etc. feel free to send me an

e-mail at peter@peterodding.com.

1.1.7 License

This software is licensed under the MIT license.

CHAPTER 2

API documentation

The following API documentation is automatically generated from the source code:

2.1 API documentation

The following documentation is based on the source code of version 8.4 of the *deb-pkg-tools* package. The following modules are available:

- deb_pkg_tools.version
- deb_pkg_tools.version.native

Note: Most of the functions defined by *deb-pkg-tools* depend on external programs. If these programs fail unexpectedly (end with a nonzero exit code) executor.ExternalCommandFailed is raised.

2.1.1 deb_pkg_tools.cache

Debian binary package metadata cache.

The *PackageCache* class implements a persistent, multiprocess cache for Debian binary package metadata. The cache supports the following binary package metadata:

- The control fields of packages;
- The files installed by packages;
- The MD5, SHA1 and SHA256 sums of packages.

The package metadata cache can speed up the following functions:

- *collect_related_packages()*
- *get_packages_entry()*
- *inspect_package()*
- *inspect_package_contents()*
- *inspect_package_fields()*
- *scan_packages()*
- *update_repository()*

Because a lot of functionality in *deb-pkg-tools* uses *inspect_package()* and its variants, the package metadata cache almost always provides a speedup compared to recalculating metadata on demand.

The cache is especially useful when you're manipulating large package repositories where relatively little metadata changes (which is a pretty common use case if you're using *deb-pkg-tools* seriously).

Internals

For several years the package metadata cache was based on SQLite and this worked fine. Then I started experimenting with concurrent builds on the same build server and I ran into SQLite raising lock timeout errors. I switched SQLite to use the Write-Ahead Log (WAL) and things seemed to improve until I experienced several corrupt databases in situations where multiple writers and multiple readers were all hitting the cache at the same time.

At this point I looked around for alternative cache backends with the following requirements:

- Support for concurrent reading and writing without any locking or blocking.
- It should not be possible to corrupt the cache, regardless of concurrency.
- To keep system requirements to a minimum, it should not be required to have a server (daemon) process running just for the cache to function.

These conflicting requirements left me with basically no options :-). Based on previous good experiences I decided to try using the filesystem to store the cache, with individual files representing cache entries. Through atomic filesystem operations this strategy basically delegates all locking to the filesystem, which should be guaranteed to do the right thing (POSIX).

Storing the cache on the filesystem like this has indeed appeared to solve all locking and corruption issues, but when the filesystem cache is cold (for example because you've just run a couple of heavy builds) it's still damn slow to scan the package metadata of a full repository with hundreds of archives...

As a pragmatic performance optimization memcached was added to the mix. Any errors involving memcached are silently ignored which means memcached isn't required to use the cache; it's an optional optimization.

deb_pkg_tools.cache.**CACHE_FORMAT_REVISION = 2**
The version number of the cache format (an integer).

deb_pkg_tools.cache.**get_default_cache**()
Load the default package cache stored inside the user's home directory.

The location of the cache is configurable using the option *package_cache_directory*, however make sure you set that option *before* calling *get_default_cache()* because the cache will be initialized only once.

Returns A *PackageCache* object.

class deb_pkg_tools.cache.**PackageCache**(*directory*)
A persistent, multiprocess cache for Debian binary package metadata.

__init__(*directory*)
Initialize a package cache.

Parameters **directory** – The pathname of the package cache directory (a string).

__getstate__()
Save a `pickle` compatible *PackageCache* representation.

The *__getstate__()* and *__setstate__()* methods make *PackageCache* objects compatible with `multiprocessing` (which uses `pickle`). This capability is used by *deb_pkg_tools.cli.collect_packages()* to enable concurrent package collection.

__setstate__(*state*)
Load a `pickle` compatible *PackageCache* representation.

connect_memcached()
Initialize a connection to the memcached daemon.

get_entry(*category*, *pathname*)
Get an object representing a cache entry.

Parameters

- **category** – The type of metadata that this cache entry represents (a string like 'control-fields', 'package-fields' or 'contents').
- **pathname** – The pathname of the package archive (a string).

Returns A *CacheEntry* object.

collect_garbage(*force=False*, *interval=86400*)
Delete any entries in the persistent cache that refer to deleted archives.

Parameters

- **force** – `True` to force a full garbage collection run (defaults to `False` which means garbage collection is performed only once per *interval*).

- **interval** – The number of seconds to delay garbage collection when *force* is `False` (a number, defaults to the equivalent of 24 hours).

class deb_pkg_tools.cache.**CacheEntry**(*cache*, *category*, *pathname*)
An entry in the package metadata cache provided by *PackageCache*.

__init__(*cache*, *category*, *pathname*)
Initialize a *CacheEntry* object.

Parameters

- **cache** – The *PackageCache* that created this entry.
- **category** – The type of metadata that this cache entry represents (a string like 'control-fields', 'package-fields' or 'contents').
- **pathname** – The pathname of the package archive (a string).

get_value()
Get the cache entry's value.

Returns A previously cached value or `None` (when the value isn't available in the cache).

set_value(*value*)
Set the cache entry's value.

Parameters **value** – The metadata to save in the cache.

set_memcached()
Helper for *get_value()* and *set_value()* to write to memcached.

up_to_date(*value*)
Helper for *get_value()* to validate cached values.

write_file(*filename*)
Helper for *set_value()* to cache values on the filesystem.

2.1.2 deb_pkg_tools.checks

Static analysis of Debian binary packages to detect common problems.

The *deb_pkg_tools.checks* module attempts to detect common problems in Debian binary package archives using static analysis. Currently there's a check that detects duplicate files in dependency sets and a check that detects version conflicts in repositories.

deb_pkg_tools.checks.**check_package**(*archive*, *cache=None*)
Perform static checks on a package's dependency set.

Parameters

- **archive** – The pathname of an existing `*.deb` archive (a string).
- **cache** – The *PackageCache* to use (defaults to `None`).

Raises *BrokenPackage* when one or more checks failed.

deb_pkg_tools.checks.**check_duplicate_files**(*dependency_set*, *cache=None*)
Check a collection of Debian package archives for conflicts.

Parameters

- **dependency_set** – A list of filenames (strings) of `*.deb` files.
- **cache** – The *PackageCache* to use (defaults to `None`).

Raises exceptions.ValueError when less than two package archives are given (the duplicate check obviously only works if there are packages to compare :-).

Raises *DuplicateFilesFound* when duplicate files are found within a group of package archives.

This check looks for duplicate files in package archives that concern different packages. Ignores groups of packages that have their 'Provides' and 'Replaces' fields set to a common value. Other variants of 'Conflicts' are not supported yet.

Because this analysis involves both the package control file fields and the pathnames of files installed by packages it can be really slow. To make it faster you can use the *PackageCache*.

deb_pkg_tools.checks.**check_version_conflicts**(*dependency_set*, *cache=None*)

Check for version conflicts in a dependency set.

Parameters

- **dependency_set** – A list of filenames (strings) of `*.deb` files.
- **cache** – The *PackageCache* to use (defaults to None).

Raises *VersionConflictFound* when one or more version conflicts are found.

For each Debian binary package archive given, check if a newer version of the same package exists in the same repository (directory). This analysis can be very slow. To make it faster you can use the *PackageCache*.

exception deb_pkg_tools.checks.**BrokenPackage**

Base class for exceptions raised by the checks defined in *deb_pkg_tools.checks*.

exception deb_pkg_tools.checks.**DuplicateFilesFound**

Raised by *check_duplicate_files()* when duplicates are found.

exception deb_pkg_tools.checks.**VersionConflictFound**

Raised by *check_version_conflicts()* when version conflicts are found.

2.1.3 deb_pkg_tools.cli

Usage: *deb-pkg-tools [OPTIONS] ...*

Wrapper for the deb-pkg-tools Python project that implements various tools to inspect, build and manipulate Debian binary package archives and related entities like trivial repositories.

Supported options:

Option	Description
-i, --inspect=FILE	Inspect the metadata in the Debian binary package archive given by FILE (similar to "dpkg --info").
-c, --collect=DIR	Copy the package archive(s) given as positional arguments (and all package archives required by the given package archives) into the directory given by DIR.
-C, --check=FILE	Perform static analysis on a package archive and its dependencies in order to recognize common errors as soon as possible.
-p, --patch=FILE	Patch fields into the existing control file given by FILE. To be used together with the -s, --set option.
-s, --set=LINE	A line to patch into the control file (syntax: "Name: Value"). To be used together with the -p, --patch option.
-b, --build=DIR	Build a Debian binary package with "dpkg-deb --build" (and lots of intermediate Python magic, refer to the API documentation of the project for full details) based on the binary package template in the directory given by DIR. The resulting archive is located in the system wide temporary directory (usually /tmp).
-u, --update-repo=DIR	Create or update the trivial Debian binary package repository in the directory given by DIR.
-a, --activate-repo=DIR	Enable "apt-get" to install packages from the trivial repository (requires root/sudo privilege) in the directory given by DIR. Alternatively you can use the -w, --with-repo option.
-d, --deactivate-repo=DIR	Cleans up after --activate-repo (requires root/sudo privilege). Alternatively you can use the -w, --with-repo option.
-w, --with-repo=DIR	Create or update a trivial package repository, activate the repository, run the positional arguments as an external command (usually "apt-get install") and finally deactivate the repository.
--gc, --garbage-collect	Force removal of stale entries from the persistent (on disk) package metadata cache. Garbage collection is performed automatically by the deb-pkg-tools command line interface when the last garbage collection cycle was more than 24 hours ago, so you only need to do it manually when you want to control when it happens (for example by a daily cron job scheduled during idle hours :-).
-y, --yes	Assume the answer to interactive questions is yes.
-v, --verbose	Make more noise! (useful during debugging)
-h, --help	Show this message and exit.

deb_pkg_tools.cli.**main**()
Command line interface for the deb-pkg-tools program.

deb_pkg_tools.cli.**show_package_metadata**(*archive*)
Show the metadata and contents of a Debian archive on the terminal.

Parameters **archive** – The pathname of an existing *.deb archive (a string).

deb_pkg_tools.cli.**highlight**(*text*)
Highlight a piece of text using ANSI escape sequences.

Parameters **text** – The text to highlight (a string).

Returns The highlighted text (when standard output is connected to a terminal) or the original text (when standard output is not connected to a terminal).

deb_pkg_tools.cli.**collect_packages**(*archives*, *directory*, *prompt=True*, *cache=None*, *concurrency=None*)
Interactively copy packages and their dependencies.

Parameters

- **archives** – An iterable of strings with the filenames of one or more `*.deb` files.
- **directory** – The pathname of a directory where the package archives and dependencies should be copied to (a string).
- **prompt** – `True` (the default) to ask confirmation from the operator (using a confirmation prompt rendered on the terminal), `False` to skip the prompt.
- **cache** – The *PackageCache* to use (defaults to `None`).
- **concurrency** – Override the number of concurrent processes (defaults to the number of *archives* given or to the value of `multiprocessing.cpu_count()`, whichever is smaller).

Raises `ValueError` when no archives are given.

When more than one archive is given a `multiprocessing` pool is used to collect related archives concurrently, in order to speed up the process of collecting large dependency sets.

deb_pkg_tools.cli.**collect_packages_worker**(*args*)
Helper for *collect_packages()* that enables concurrent collection.

deb_pkg_tools.cli.**smart_copy**(*src*, *dst*)
Create a hard link to or copy of a file.

Parameters

- **src** – The pathname of the source file (a string).
- **dst** – The pathname of the target file (a string).

This function first tries to create a hard link *dst* pointing to *src* and if that fails it will perform a regular file copy from *src* to *dst*. This is used by *collect_packages()* in an attempt to conserve disk space when copying package archives between repositories on the same filesystem.

deb_pkg_tools.cli.**with_repository_wrapper**(*directory*, *command*, *cache*)
Command line wrapper for *deb_pkg_tools.repo.with_repository()*.

Parameters

- **directory** – The pathname of a directory with `*.deb` archives (a string).
- **command** – The command to execute (a list of strings).
- **cache** – The *PackageCache* to use (defaults to `None`).

deb_pkg_tools.cli.**check_directory**(*argument*)
Make sure a command line argument points to an existing directory.

Parameters **argument** – The original command line argument.

Returns The absolute pathname of an existing directory.

deb_pkg_tools.cli.**say**(*text*, **args*, ***kw*)
Reliably print Unicode strings to the terminal (standard output stream).

2.1.4 deb_pkg_tools.config

Configuration defaults for the *deb-pkg-tools* package.

deb_pkg_tools.config.**system_config_directory = '/etc/deb-pkg-tools'**
The pathname of the global (system wide) configuration directory used by *deb-pkg-tools* (a string).

deb_pkg_tools.config.**system_cache_directory = '/var/cache/deb-pkg-tools'**
The pathname of the global (system wide) package cache directory (a string).

deb_pkg_tools.config.**user_config_directory = '/home/docs/.deb-pkg-tools'**
The pathname of the current user's configuration directory used by *deb-pkg-tools* (a string).

Default The expanded value of `~/.deb-pkg-tools`.

deb_pkg_tools.config.**user_cache_directory = '/home/docs/.cache/deb-pkg-tools'**
The pathname of the current user's package cache directory (a string).

Default The expanded value of `~/.cache/deb-pkg-tools`.

deb_pkg_tools.config.**package_cache_directory = '/home/docs/.cache/deb-pkg-tools'**
The pathname of the selected package cache directory (a string).

Default The value of *system_cache_directory* when running as `root`, the value of *user_cache_directory* otherwise.

deb_pkg_tools.config.**repo_config_file = 'repos.ini'**
The base name of the configuration file with user-defined Debian package repositories (a string).

This configuration file is loaded from *system_config_directory* and/or *user_config_directory*.

Default The string `repos.ini`.

2.1.5 deb_pkg_tools.control

Functions to manipulate Debian control files.

The functions in the *deb_pkg_tools.control* module can be used to manipulate Debian control files. It was developed specifically for control files of binary packages, however the code is very generic.

This module makes extensive use of case insensitivity provided by the `humanfriendly.case` module:

- The dictionaries returned by this module are case insensitive.
- The enumerations *MANDATORY_BINARY_CONTROL_FIELDS* and *DEPENDS_LIKE_FIELDS* contain case insensitive strings.

Case insensitivity was originally added to this module by virtue of its integration with python-debian. Since then this dependency was removed but the case insensitive behavior was preserved for the sake of backwards compatibility.

Note: Deprecated names

The following aliases exist to preserve backwards compatibility, however a `DeprecationWarning` is triggered when they are accessed, because these aliases will be removed in a future release.

deb_pkg_tools.control.**deb822_from_string**
Alias for *deb_pkg_tools.deb822.parse_deb822*.

deb_pkg_tools.control.**Deb822**
Alias for *deb_pkg_tools.deb822.Deb822*.

---

deb_pkg_tools.control.**MANDATORY_BINARY_CONTROL_FIELDS = (u'Architecture', u'Description', u**
A tuple of strings (actually `CaseInsensitiveKey` objects) with the canonical names of the mandatory binary control file fields as defined by the Debian policy manual.

deb_pkg_tools.control.**DEFAULT_CONTROL_FIELDS = {u'Architecture': 'all', u'Priority': 'opt**
A case insensitive dictionary with string key/value pairs. Each key is the canonical name of a binary control file field and each value is the default value given to that field by *create_control_file()* when the caller hasn't defined a value for the field.

deb_pkg_tools.control.**DEPENDS_LIKE_FIELDS = (u'Breaks', u'Conflicts', u'Depends', u'Enhance**
A tuple of strings with the canonical names of control file fields that are similar to the `Depends` field (in the sense that they contain a comma separated list of package names with optional version specifications).

deb_pkg_tools.control.**SPECIAL_CASES = {'md5sum': 'MD5sum', 'sha1': 'SHA1', 'sha256': 'SH**
A dictionary with string key/value pairs of non-default casing for words that are part of control field names. The keys are intentionally normalized to lowercase, whereas the values contain the proper casing. Used by *normalize_control_field_name()*.

deb_pkg_tools.control.**load_control_file**(*control_file*)
Load a control file and return the parsed control fields.

Parameters **control_file** – The filename of the control file to load (a string).

Returns A dictionary created by *parse_control_fields()*.

deb_pkg_tools.control.**create_control_file**(*control_file*, *control_fields*)
Create a Debian control file.

Parameters

- **control_file** – The filename of the control file to create (a string).
- **control_fields** – A dictionary with control file fields. This dictionary is merged with the values in *DEFAULT_CONTROL_FIELDS*.

Raises See *check_mandatory_fields()*.

deb_pkg_tools.control.**check_mandatory_fields**(*control_fields*)
Make sure mandatory binary control fields are defined.

Parameters **control_fields** – A dictionary with control file fields.

Raises `ValueError` when a mandatory binary control field is not present in the provided control fields (see also *MANDATORY_BINARY_CONTROL_FIELDS*).

deb_pkg_tools.control.**patch_control_file**(*control_file*, *overrides*)
Patch the fields of a Debian control file.

Parameters

- **control_file** – The filename of the control file to patch (a string).
- **overrides** – A dictionary with fields that should override default name/value pairs. Values of the fields *Depends*, *Provides*, *Replaces* and *Conflicts* are merged while values of other fields are overwritten.

deb_pkg_tools.control.**merge_control_fields**(*defaults*, *overrides*)
Merge the fields of two Debian control files.

Parameters

- **defaults** – A dictionary with existing control field name/value pairs.
- **overrides** – A dictionary with fields that should override default name/value pairs. Values of the fields *Depends*, *Provides*, *Replaces* and *Conflicts* are merged while values of other fields are overwritten.

Returns A dictionary of the type *Deb822*.

deb_pkg_tools.control.**parse_control_fields**(*input_fields*)

Parse Debian control file fields.

Parameters **input_fields** – The dictionary to convert.

Returns A dictionary of the type *Deb822*.

This function takes the result of the shallow parsing of control fields performed by *parse_deb822()* and massages the data into a friendlier format:

- The values of the fields given by *DEPENDS_LIKE_FIELDS* are parsed into Python data structures using *parse_depends()*.
- The value of the *Installed-Size* field is converted to an integer.

Let's look at an example. We start with the raw control file contents so you can see the complete input:

```
>>> from deb_pkg_tools.deb822 import parse_deb822
>>> unparsed_fields = parse_deb822('''
... Package: python3.4-minimal
... Version: 3.4.0-1+precise1
... Architecture: amd64
... Installed-Size: 3586
... Pre-Depends: libc6 (>= 2.15)
... Depends: libpython3.4-minimal (= 3.4.0-1+precise1), libexpat1 (>= 1.95.8),␣
↪libgcc1 (>= 1:4.1.1), zlib1g (>= 1:1.2.0), foo | bar
... Recommends: python3.4
... Suggests: binfmt-support
... Conflicts: binfmt-support (<< 1.1.2)
... ''')
```

Here are the control file fields as parsed by *parse_deb822()*:

```
>>> print(repr(unparsed_fields))
{'Architecture': u'amd64',
 'Conflicts': u'binfmt-support (<< 1.1.2)',
 'Depends': u'libpython3.4-minimal (= 3.4.0-1+precise1), libexpat1 (>= 1.95.8),␣
↪libgcc1 (>= 1:4.1.1), zlib1g (>= 1:1.2.0), foo | bar',
 'Installed-Size': u'3586',
 'Package': u'python3.4-minimal',
 'Pre-Depends': u'libc6 (>= 2.15)',
 'Recommends': u'python3.4',
 'Suggests': u'binfmt-support',
 'Version': u'3.4.0-1+precise1'}
```

Notice the value of the *Depends* line is a comma separated string, i.e. it hasn't been parsed. Now here are the control file fields parsed by the *parse_control_fields()* function:

```
>>> from deb_pkg_tools.control import parse_control_fields
>>> parsed_fields = parse_control_fields(unparsed_fields)
>>> print(repr(parsed_fields))
{'Architecture': u'amd64',
 'Conflicts': RelationshipSet(VersionedRelationship(name=u'binfmt-support',␣
↪operator=u'<<', version=u'1.1.2')),
 'Depends': RelationshipSet(VersionedRelationship(name=u'libpython3.4-minimal',␣
↪operator=u'=', version=u'3.4.0-1+precise1'),
                            VersionedRelationship(name=u'libexpat1', operator=u'>=
↪', version=u'1.95.8'),
                            VersionedRelationship(name=u'libgcc1', operator=u'>=',
↪ version=u'1:4.1.1'),
```

(continues on next page)

(continued from previous page)

```
                          VersionedRelationship(name=u'zlib1g', operator=u'>=',␣
↪version=u'1:1.2.0'),
                          AlternativeRelationship(Relationship(name=u'foo'),␣
↪Relationship(name=u'bar'))),
 'Installed-Size': 3586,
 'Package': u'python3.4-minimal',
 'Pre-Depends': RelationshipSet(VersionedRelationship(name=u'libc6', operator=u'>=
↪', version=u'2.15')),
 'Recommends': u'python3.4',
 'Suggests': RelationshipSet(Relationship(name=u'binfmt-support')),
 'Version': u'3.4.0-1+precise1'}
```

For more information about fields like *Depends* and *Suggests* please refer to the documentation of *parse_depends()*.

deb_pkg_tools.control.**unparse_control_fields**(*input_fields*)
Unparse (undo the parsing of) Debian control file fields.

Parameters **input_fields** – A dict object previously returned by *parse_control_fields()*.

Returns A dictionary of the type *Deb822*.

This function converts dictionaries created by *parse_control_fields()* back into shallow dictionaries of strings. Fields with an empty value are omitted. This makes it possible to delete fields from a control file with *patch_control_file()* by setting the value of a field to None in the overrides...

deb_pkg_tools.control.**normalize_control_field_name**(*name*)
Normalize the case of a field name in a Debian control file.

Parameters **name** – The name of a control file field (a string).

Returns The normalized name (a string of the type CaseInsensitiveKey).

Normalization of control file field names is useful to simplify control file manipulation and in particular the merging of control files.

According to the Debian Policy Manual (section 5.1, Syntax of control files) field names are not case-sensitive, however in my experience deviating from the standard capitalization can break things. Hence this function (which is used by the other functions in the *deb_pkg_tools.control* module).

Note: This function doesn't adhere 100% to the Debian policy because it lacks special casing (no pun intended ;-) for fields like DM-Upload-Allowed. It's not clear to me if this will ever become a relevant problem for building simple binary packages... (which explains why I didn't bother to implement special casing)

2.1.6 deb_pkg_tools.deb822

Parsing and formatting of Debian control fields in the deb822 format.

deb_pkg_tools.deb822.**dump_deb822**(*fields*)
Format the given Debian control fields as text.

Parameters **fields** – The control fields to dump (a dictionary).

Returns A Unicode string containing the formatted control fields.

deb_pkg_tools.deb822.**parse_deb822**(*text*, *filename=None*)
Parse Debian control fields into a *Deb822* object.

Parameters

- **text** – A string containing the control fields to parse.
- **filename** – An optional string with the filename of the source file from which the control fields were extracted (only used for the purpose of error reporting).

Returns A *Deb822* object.

class deb_pkg_tools.deb822.**Deb822**(*other=None*, ***kw*)

Case insensitive dictionary to represent the fields of a parsed deb822 paragraph.

This class imitates the class of the same name in the python-debian package, primarily in the form of the *dump()* method, however that's also where the similarities end (full compatibility is not a goal).

dump(*handle=None*)

Dump the control fields to a file.

Parameters **handle** – A file-like object or None.

Returns If *handle* is None the dumped control fields are returned as a Unicode string.

__eq__(*other*)

Compare two *Deb822* objects while ignoring differences in the order of keys.

2.1.7 deb_pkg_tools.deps

Parsing and evaluation of Debian package relationship declarations.

The *deb_pkg_tools.deps* module provides functions to parse and evaluate Debian package relationship declarations as defined in chapter 7 of the Debian policy manual. The most important function is *parse_depends()* which returns a *RelationshipSet* object. The *RelationshipSet.matches()* method can be used to evaluate relationship expressions. The relationship parsing is implemented in pure Python (no external dependencies) but relationship evaluation uses the external command `dpkg --compare-versions` to ensure compatibility with Debian's package version comparison algorithm.

To give you an impression of how to use this module:

```
>>> from deb_pkg_tools.deps import parse_depends
>>> dependencies = parse_depends('python (>= 2.6), python (<< 3) | python (>= 3.4)')
>>> dependencies.matches('python', '2.5')
False
>>> dependencies.matches('python', '3.0')
False
>>> dependencies.matches('python', '2.6')
True
>>> dependencies.matches('python', '3.4')
True
>>> print(repr(dependencies))
RelationshipSet(VersionedRelationship(name='python', operator='>=', version='2.6',␣
↪architectures=()),
                AlternativeRelationship(VersionedRelationship(name='python', operator=
↪'<<', version='3', architectures=()),
                                        VersionedRelationship(name='python', operator=
↪'>=', version='3.4', architectures=())))
>>> print(str(dependencies))
python (>= 2.6), python (<< 3) | python (>= 3.4)
```

As you can see the `repr()` output of the relationship set shows the object tree and the `str` output is the dependency line.

deb_pkg_tools.deps.**parse_depends**(*relationships*)

Parse a Debian package relationship declaration line.

Parameters **relationships** – A string containing one or more comma separated package relationships or a list of strings with package relationships.

Returns A *RelationshipSet* object.

Raises ValueError when parsing fails.

This function parses a list of package relationships of the form `python (>= 2.6), python (<< 3)`, i.e. a comma separated list of relationship expressions. Uses *parse_alternatives()* to parse each comma separated expression.

Here's an example:

```
>>> from deb_pkg_tools.deps import parse_depends
>>> dependencies = parse_depends('python (>= 2.6), python (<< 3)')
>>> print(repr(dependencies))
RelationshipSet(VersionedRelationship(name='python', operator='>=', version='2.6
↪'),
                VersionedRelationship(name='python', operator='<<', version='3'))
>>> dependencies.matches('python', '2.5')
False
>>> dependencies.matches('python', '2.6')
True
>>> dependencies.matches('python', '2.7')
True
>>> dependencies.matches('python', '3.0')
False
```

deb_pkg_tools.deps.**parse_alternatives**(*expression*)

Parse an expression containing one or more alternative relationships.

Parameters **expression** – A relationship expression (a string).

Returns A *Relationship* object.

Raises ValueError when parsing fails.

This function parses an expression containing one or more alternative relationships of the form `python2.6 | python2.7`., i.e. a list of relationship expressions separated by `|` tokens. Uses *parse_relationship()* to parse each `|` separated expression.

An example:

```
>>> from deb_pkg_tools.deps import parse_alternatives
>>> parse_alternatives('python2.6')
Relationship(name='python2.6')
>>> parse_alternatives('python2.6 | python2.7')
AlternativeRelationship(Relationship(name='python2.6'),
                        Relationship(name='python2.7'))
```

deb_pkg_tools.deps.**parse_relationship**(*expression*)

Parse an expression containing a package name and optional version/architecture restrictions.

Parameters **expression** – A relationship expression (a string).

Returns A *Relationship* object.

Raises ValueError when parsing fails.

This function parses relationship expressions containing a package name and (optionally) a version relation of the form `python (>= 2.6)` and/or an architecture restriction (refer to the Debian policy manual's documentation on the syntax of relationship fields for details). Here's an example:

```
>>> from deb_pkg_tools.deps import parse_relationship
>>> parse_relationship('python')
Relationship(name='python')
>>> parse_relationship('python (<< 3)')
VersionedRelationship(name='python', operator='<<', version='3')
```

deb_pkg_tools.deps.**cache_matches**(*f*)

High performance memoizing decorator for overrides of *Relationship.matches()*.

Before writing this function I tried out several caching decorators from PyPI, unfortunately all of them were bloated. I benchmarked using *collect_related_packages()* and where this decorator would get a total runtime of 8 seconds the other caching decorators would get something like 40 seconds...

class deb_pkg_tools.deps.**AbstractRelationship**(***kw*)

Abstract base class for the various types of relationship objects defined in *deb_pkg_tools.deps*.

names

The name(s) of the packages in the relationship.

Returns A set of package names (strings).

Note: This property needs to be implemented by subclasses.

matches(*name*, *version=None*)

Check if the relationship matches a given package and version.

Parameters

- **name** – The name of a package (a string).
- **version** – The version number of a package (a string, optional).

Returns

One of the values `True`, `False` or `None` meaning the following:

- `True` if the name matches and the version doesn't invalidate the match,
- `False` if the name matches but the version invalidates the match,
- `None` if the name doesn't match at all.

Note: This method needs to be implemented by subclasses.

class deb_pkg_tools.deps.**Relationship**(***kw*)

A simple package relationship referring only to the name of a package.

Created by *parse_relationship()*.

name

The name of a package (a string).

Note: The *name* property is a `key_property`. You are required to provide a value for this property by calling the constructor of the class that defines the property with a keyword argument named *name* (unless

a custom constructor is defined, in this case please refer to the documentation of that constructor). Once this property has been assigned a value you are not allowed to assign a new value to the property.

architectures
The architecture restriction(s) on the relationship (a tuple of strings).

Note: The *architectures* property is a key_property. You are required to provide a value for this property by calling the constructor of the class that defines the property with a keyword argument named *architectures* (unless a custom constructor is defined, in this case please refer to the documentation of that constructor). Once this property has been assigned a value you are not allowed to assign a new value to the property.

names
The name(s) of the packages in the relationship.

matches(*name*, *version=None*)
Check if the relationship matches a given package name.

Parameters

- **name** – The name of a package (a string).
- **version** – The version number of a package (this parameter is ignored).

Returns True if the name matches, None otherwise.

Raises NotImplementedError when *architectures* is not empty (because evaluation of architecture restrictions hasn't been implemented).

__repr__()
Serialize a *Relationship* object to a Python expression.

__unicode__()
Serialize a *Relationship* object to a Debian package relationship expression.

class deb_pkg_tools.deps.**VersionedRelationship**(***kw*)
A conditional package relationship that refers to a package and certain versions of that package.

Created by *parse_relationship()*.

operator
An operator that compares Debian package version numbers (a string).

Note: The *operator* property is a key_property. You are required to provide a value for this property by calling the constructor of the class that defines the property with a keyword argument named *operator* (unless a custom constructor is defined, in this case please refer to the documentation of that constructor). Once this property has been assigned a value you are not allowed to assign a new value to the property.

version
The version number of a package (a string).

Note: The *version* property is a key_property. You are required to provide a value for this property by calling the constructor of the class that defines the property with a keyword argument named *version*

(unless a custom constructor is defined, in this case please refer to the documentation of that constructor). Once this property has been assigned a value you are not allowed to assign a new value to the property.

---

matches(*package*, *version=None*)
Check if the relationship matches a given package name and version.

Parameters

- **name** – The name of a package (a string).
- **version** – The version number of a package (a string, optional).

Returns

One of the values `True`, `False` or `None` meaning the following:

- `True` if the name matches and the version doesn't invalidate the match,
- `False` if the name matches but the version invalidates the match,
- `None` if the name doesn't match at all.

Raises `NotImplementedError` when *architectures* is not empty (because evaluation of architecture restrictions hasn't been implemented).

Uses the external command `dpkg --compare-versions` to ensure compatibility with Debian's package version comparison algorithm.

__repr__()
Serialize a *VersionedRelationship* object to a Python expression.

__unicode__()
Serialize a *VersionedRelationship* object to a Debian package relationship expression.

class deb_pkg_tools.deps.**AlternativeRelationship**(**relationships*)
A package relationship that refers to one of several alternative packages.

Created by *parse_alternatives()*.

__init__(**relationships*)
Initialize an *AlternativeRelationship* object.

Parameters **relationships** – One or more *Relationship* objects.

relationships
A tuple of *Relationship* objects.

Note: The *relationships* property is a `key_property`. You are required to provide a value for this property by calling the constructor of the class that defines the property with a keyword argument named *relationships* (unless a custom constructor is defined, in this case please refer to the documentation of that constructor). Once this property has been assigned a value you are not allowed to assign a new value to the property.

---

names
Get the name(s) of the packages in the alternative relationship.

Returns A set of package names (strings).

matches(*package*, *version=None*)
Check if the relationship matches a given package and version.

Parameters

- **name** – The name of a package (a string).
- **version** – The version number of a package (a string, optional).

Returns `True` if the name and version of an alternative match, `False` if the name of an alternative was matched but the version didn't match, `None` otherwise.

__repr__()
Serialize an *AlternativeRelationship* object to a Python expression.

__unicode__()
Serialize an *AlternativeRelationship* object to a Debian package relationship expression.

class deb_pkg_tools.deps.**RelationshipSet**(**relationships*)
A set of package relationships. Created by *parse_depends()*.

__init__(**relationships*)
Initialize a :class *RelationshipSet* object.

Parameters **relationships** – One or more *Relationship* objects.

relationships
A tuple of *Relationship* objects.

Note: The *relationships* property is a `key_property`. You are required to provide a value for this property by calling the constructor of the class that defines the property with a keyword argument named *relationships* (unless a custom constructor is defined, in this case please refer to the documentation of that constructor). Once this property has been assigned a value you are not allowed to assign a new value to the property.

---

names
Get the name(s) of the packages in the relationship set.

Returns A set of package names (strings).

matches(*package*, *version=None*)
Check if the set of relationships matches a given package and version.

Parameters

- **name** – The name of a package (a string).
- **version** – The version number of a package (a string, optional).

Returns `True` if all matched relationships evaluate to true, `False` if a relationship is matched and evaluates to false, `None` otherwise.

Warning: Results are cached in the assumption that *RelationshipSet* objects are immutable. This is not enforced.

__repr__(*pretty=False*, *indent=0*)
Serialize a *RelationshipSet* object to a Python expression.

__unicode__()
Serialize a *RelationshipSet* object to a Debian package relationship expression.

__iter__()
Iterate over the relationships in a relationship set.

2.1.8 deb_pkg_tools.gpg

GPG key pair generation and signing of `Release` files.

The *deb_pkg_tools.gpg* module is used to manage GPG key pairs. It allows callers to specify which GPG key pair and/or key ID they want to use and will automatically generate GPG key pairs that don't exist yet.

GnuPG 2.1 compatibility

In 2018 the *deb_pkg_tools.gpg* module got a major update to enable compatibility with GnuPG >= 2.1:

- The *deb_pkg_tools.gpg* module was first integrated into deb-pkg-tools in 2013 and was developed based on GnuPG 1.4.10 which was the version included in Ubuntu 10.04.
- Ubuntu 18.04 includes GnuPG 2.2.4 which differs from 1.4.10 in several backwards incompatible ways that require changes in deb-pkg-tools which directly affect the users of deb-pkg-tools (the API has changed).

The following sections discuss the concrete changes:

Storage of secret keys

The storage of secret keys has changed in a backwards incompatible way, such that the `--secret-keyring` command line option is now obsolete and ignored. The GnuPG documentation suggests to use an ephemeral home directory as a replacement for `--secret-keyring`. To enable compatibility with GnuPG >= 2.1 while at the same time preserving compatibility with older releases, the *GPGKey* class gained a new *directory* property:

- When GnuPG >= 2.1 is detected *directory* is required.
- When GnuPG < 2.1 is detected *directory* may be specified and will be respected, but you can also use "the old calling convention" where the *public_key_file*, *secret_key_file* and *key_id* properties are specified separately.
- The documentation of the *GPGKey* initializer explains how to enable compatibility with old and new versions GnuPG versions at the same time (using the same Python code).

Unattended key generation

The default behavior of `gpg --batch --gen-key` has changed:

- The user is now presented with a GUI prompt that asks to specify a pass phrase for the new key, at which point the supposedly unattended key generation is effectively blocked on user input...
- To avoid the GUI prompt the new `%no-protection` option needs to be added to the batch file, but of course that option will not be recognized by older GnuPG releases, so it needs to be added conditionally.

deb_pkg_tools.gpg.**FORCE_ENTROPY = False**
`True` to allow *GPGKey.generate_key_pair()* to force the system to generate entropy based on disk I/O , `False` to disallow this behavior (the default).

This was added to facilitate the deb-pkg-tools test suite running on Travis CI. It is assumed that this rather obscure functionality will only ever be useful in the same context: Running a test suite in a virtualization environment with very low entropy.

The environment variable `$DPT_FORCE_ENTROPY` can be used to control the value of this variable (see `coerce_boolean()` for acceptable values).

deb_pkg_tools.gpg.**GPG_AGENT_VARIABLE = 'GPG_AGENT_INFO'**
The name of the environment variable used to communicate between the GPG agent and gpg processes (a string).

deb_pkg_tools.gpg.**create_directory**(*pathname*)
Create a GnuPG directory with sane permissions (to avoid GnuPG warnings).

Parameters **pathname** – The directory to create (a string).

deb_pkg_tools.gpg.**have_updated_gnupg**()
Check which version of GnuPG is installed.

Returns `True` if GnuPG >= 2.1 is installed, `False` for older versions.

deb_pkg_tools.gpg.**initialize_gnupg**()
Make sure the `~/.gnupg` directory exists.

Older versions of GPG can/will fail when the `~/.gnupg` directory doesn't exist (e.g. in a newly created chroot). GPG itself creates the directory after noticing that it's missing, but then still fails! Later runs work fine however. To avoid this problem we make sure `~/.gnupg` exists before we run GPG.

class deb_pkg_tools.gpg.**GPGKey**(***options*)
Container for generating GPG key pairs and signing release files.

This class is used to sign `Release` files in Debian package repositories. If the given GPG key pair doesn't exist yet it will be automatically created without user interaction (except gathering of entropy, which is not something I can automate :-).

__init__(***options*)
Initialize a *GPGKey* object.

Parameters **options** – Refer to the initializer of the superclass (`PropertyManager`) for details about argument handling.

There are two ways to specify the location of a GPG key pair:

- The old way applies to GnuPG < 2.1 and uses *public_key_file* and *secret_key_file*.
- The new way applies to GnuPG >= 2.1 and uses *directory*.

If you don't specify anything the user's default key pair will be used. Specifying all three properties enables isolation from the user's default keyring that's compatible with old and new GnuPG installations at the same time.

You can also use *key_id* to select a specific existing GPG key pair, possibly in combination with the previously mentioned properties.

When the caller has specified a custom location for the GPG key pair but the associated files don't exist yet a new GPG key pair will be automatically generated. This requires that *name* and *description* have been set.

check_key_id()
Raise `EnvironmentError` when a key ID has been specified but the key pair doesn't exist.

check_new_usage()
Raise an exception when detecting a backwards incompatibility.

Raises `TypeError` as described below.

When GnuPG >= 2.1 is installed the *check_new_usage()* method is called to make sure that the caller is aware of the changes in API contract that this implies. We do so by raising an exception when both of the following conditions hold:

- The caller is using the old calling convention of setting *public_key_file* and *secret_key_file* (which confirms that the intention is to use an isolated GPG key).
- The caller is not using the new calling convention of setting *directory* (even though this is required to use an isolated GPG key with GnuPG >= 2.1).

check_old_files()
Raise an exception when we risk overwriting an existing public or secret key file.

Returns A list of filenames with existing files.

Raises EnvironmentError as described below.

When GnuPG < 2.1 is installed *check_old_files()* is called to ensure that when *public_key_file* and *secret_key_file* have been provided, either both of the files already exist or neither one exists. This avoids accidentally overwriting an existing file that wasn't generated by deb-pkg-tools and shouldn't be touched at all.

check_old_usage()
Raise an exception when either the public or the secret key hasn't been provided.

Raises TypeError as described below.

When GnuPG < 2.1 is installed *check_old_usage()* is called to ensure that *public_key_file* and *secret_key_file* are either both provided or both omitted.

generate_key_pair()
Generate a missing GPG key pair on demand.

Raises TypeError when the GPG key pair needs to be generated (because it doesn't exist yet) but no *name* and *description* were provided.

set_old_defaults()
Fall back to the default public and secret key files for GnuPG < 2.1.

batch_script
A GnuPG batch script suitable for `gpg --batch --gen-key` (a string).

Note: The *batch_script* property is a cached_property. This property's value is computed once (the first time it is accessed) and the result is cached. To clear the cached value you can use del or delattr().

---

command_name
The name of the GnuPG program (a string, defaults to gpg).

Note: The *command_name* property is a mutable_property. You can change the value of this property using normal attribute assignment syntax. To reset it to its default (computed) value you can use del or delattr().

---

description
The description of the GPG key pair (a string or None).

Used only when the key pair is generated because it doesn't exist yet.

Note: The *description* property is a mutable_property. You can change the value of this property using normal attribute assignment syntax. To reset it to its default (computed) value you can use del or delattr().

directory
The pathname of the GnuPG home directory to use (a string or None).

This property was added in deb-pkg-tools 5.0 to enable compatibility with GnuPG >= 2.1 which changed the storage of secret keys in a backwards incompatible way by obsoleting the --secret-keyring command line option. The GnuPG documentation suggests to use an ephemeral home directory as a replacement and that's why the *directory* property was added.

Note: The *directory* property is a mutable_property. You can change the value of this property using normal attribute assignment syntax. To reset it to its default (computed) value you can use del or delattr().

directory_default
The pathname of the default GnuPG home directory (a string).

Note: The *directory_default* property is a cached_property. This property's value is computed once (the first time it is accessed) and the result is cached. To clear the cached value you can use del or delattr().

directory_effective
The pathname of the GnuPG home directory that will actually be used (a string).

Note: The *directory_effective* property is a cached_property. This property's value is computed once (the first time it is accessed) and the result is cached. To clear the cached value you can use del or delattr().

existing_files
A list of strings with the filenames of existing GnuPG data files.

The content of this list depends on the GnuPG version:

- On GnuPG >= 2.1 and/or when *directory* has been set (also on GnuPG < 2.1) any files in or below *directory* are included.
- On GnuPG < 2.1 *public_key_file* and *secret_key_file* are included (only if the properties are set and the files exist of course).

Note: The *existing_files* property is a cached_property. This property's value is computed once (the first time it is accessed) and the result is cached. To clear the cached value you can use del or delattr().

identifier
A unique identifier for the GPG key pair (a string).

The output of the gpg --list-keys --with-colons command is parsed to extract a unique identifier for the GPG key pair:

- When a fingerprint is available this is preferred.
- Otherwise a long key ID will be returned (assuming one is available).
- If neither can be extracted `EnvironmentError` is raised.

If an isolated key pair is being used the *directory* option should be used instead of the *public_key_file* and *secret_key_file* properties, even if GnuPG < 2.1 is being used. This is necessary because of what appears to be a bug in GnuPG, see this mailing list thread for more discussion.

Note: The *identifier* property is a `cached_property`. This property's value is computed once (the first time it is accessed) and the result is cached. To clear the cached value you can use `del` or `delattr()`.

---

gpg_command
The GPG command line that can be used to sign using the key, export the key, etc (a string).

The value of *gpg_command* is based on *scoped_command* combined with the `--no-default-keyring`

The documentation of *GPGKey.__init__()* contains two examples.

key_id
The key ID of an existing key pair to use (a string or `None`).

If this option is provided then the key pair must already exist.

Note: The *key_id* property is a `mutable_property`. You can change the value of this property using normal attribute assignment syntax. To reset it to its default (computed) value you can use `del` or `delattr()`.

---

name
The name of the GPG key pair (a string or `None`).

Used only when the key pair is generated because it doesn't exist yet.

Note: The *name* property is a `mutable_property`. You can change the value of this property using normal attribute assignment syntax. To reset it to its default (computed) value you can use `del` or `delattr()`.

---

new_usage
`True` if the new API is being used, `False` otherwise.

old_usage
`True` if the old API is being used, `False` otherwise.

public_key_file
The pathname of the public key file (a string or `None`).

This is only used when GnuPG < 2.1 is installed.

Note: The *public_key_file* property is a `mutable_property`. You can change the value of this property using normal attribute assignment syntax. To reset it to its default (computed) value you can use `del` or `delattr()`.

---

scoped_command
The GPG program name and optional `--homedir` command line option (a list of strings).

The name of the GPG program is taken from *command_name* and the `--homedir` option is only added when *directory* is set.

secret_key_file
The pathname of the secret key file (a string or `None`).

This is only used when GnuPG < 2.1 is installed.

Note: The *secret_key_file* property is a `mutable_property`. You can change the value of this property using normal attribute assignment syntax. To reset it to its default (computed) value you can use `del` or `delattr()`.

---

use_agent
Whether to enable the use of the GPG agent (a boolean).

This property checks whether the environment variable given by *GPG_AGENT_VARIABLE* is set to a nonempty value. If it is then *gpg_command* will include the `--use-agent` option. This makes it possible to integrate repository signing with the GPG agent, so that a password is asked for once instead of every time something is signed.

class deb_pkg_tools.gpg.**EntropyGenerator**
Force the system to generate entropy based on disk I/O.

The *deb-pkg-tools* test suite runs on Travis CI which uses virtual machines to isolate tests. Because the *deb-pkg-tools* test suite generates several GPG keys it risks the chance of getting stuck and being killed after 10 minutes of inactivity. This happens because of a lack of entropy which is a very common problem in virtualized environments. There are tricks to use fake entropy to avoid this problem:

- The *rng-tools* package/daemon can feed `/dev/random` based on `/dev/urandom`. Unfortunately this package doesn't work on Travis CI because they use OpenVZ which uses read only `/dev/random` devices.
- GPG version 2 supports the `--debug-quick-random` option but I haven't investigated how easy it is to switch.

Instances of this class can be used as a context manager to generate endless disk I/O which is one of the few sources of entropy on virtualized systems. Entropy generation is enabled when the environment variable `$DPT_FORCE_ENTROPY` is set to `yes`, `true` or `1`.

__init__()
Initialize a *EntropyGenerator* object.

__enter__()
Enable entropy generation.

__exit__(*exc_type*, *exc_value*, *traceback*)
Disable entropy generation.

deb_pkg_tools.gpg.**generate_entropy**()
Force the system to generate entropy based on disk I/O.

This function is run in a separate process by *EntropyGenerator*. It scans the complete file system and reads every file it finds in blocks of 1 KB. This function never returns; it has to be killed.

2.1.9 deb_pkg_tools.package

Functions to build and inspect Debian binary package archives (`*.deb` files).

deb_pkg_tools.package.**BINARY_PACKAGE_ARCHIVE_EXTENSIONS = ('.deb', '.udeb')**
A tuple of strings with supported filename extensions of Debian binary package archives. Used by *find_package_archives()* and *parse_filename()*.

deb_pkg_tools.package.**DEPENDENCY_FIELDS = ('Depends', 'Pre-Depends')**
A tuple of strings with names of control file fields that specify dependencies, used by *collect_related_packages()* to analyze dependency trees.

deb_pkg_tools.package.**DIRECTORIES_TO_REMOVE = ('.bzr', '.git', '.hg', '.svn', '__pycache__**
A tuple of strings with `fnmatch` patterns of directories to remove before building a package. Used by *clean_package_tree()* which is called by *build_package()*. Avoids the following Lintian warnings:

- package-contains-vcs-control-dir
- package-installs-python-pycache-dir

deb_pkg_tools.package.**FILES_TO_REMOVE = ('*.pyc', '*.pyo', '*~', '.*.s??', '.DS_Store', '.**
A tuple of strings with `fnmatch` patterns of files to remove before building a package. Used by *clean_package_tree()* which is called by *build_package()*. Avoids the following Lintian warnings:

- backup-file-in-package
- macos-ds-store-file-in-package
- macos-resource-fork-file-in-package
- package-contains-vcs-control-file
- package-installs-python-bytecode

deb_pkg_tools.package.**OBJECT_FILE_EXCLUDES = ('*.eot', '*.gif', '*.ico', '*.jpeg', '*.jpg',**
A tuple of strings with `fnmatch` patterns of common file types to be ignored by *find_object_files()* even if the files in question have the executable bit set and contain binary data.

This option was added to minimize harmless but possibly confusing warnings from *strip_object_files()* and/or *find_system_dependencies()* caused by binary files that happen to (incorrectly) have their executable bit set.

deb_pkg_tools.package.**ALLOW_CHOWN = True**
`True` to allow *build_package()* to normalize file ownership by running chown, `False` to disallow usage of chown.

The environment variable `$DPT_CHOWN_FILES` can be used to control the value of this variable (see `coerce_boolean()` for acceptable values).

deb_pkg_tools.package.**ALLOW_FAKEROOT_OR_SUDO = True**
`True` to allow *build_package()* to use fakeroot (when available) or sudo (when fakeroot is not available), `False` to disallow this behavior.

The environment variable `$DPT_ALLOW_FAKEROOT_OR_SUDO` can be used to control the value of this variable (see `coerce_boolean()` for acceptable values).

deb_pkg_tools.package.**ALLOW_HARD_LINKS = True**
`True` to allow *copy_package_files()* to use hard links to optimize file copying, `False` to disallow this behavior.

The environment variable `$DPT_HARD_LINKS` can be used to control the value of this variable (see `coerce_boolean()` for acceptable values).

deb_pkg_tools.package.**ALLOW_RESET_SETGID = True**
`True` to allow *`build_package()`* to remove the sticky bit from directories, `False` to disallow this behavior.

The environment variable `$DPT_RESET_SETGID` can be used to control the value of this variable (see `coerce_boolean()` for acceptable values).

deb_pkg_tools.package.**PARSE_STRICT = True**
If *`PARSE_STRICT`* is `True` then *`parse_filename()`* expects filenames of `*.deb` archives to encode the package name, version and architecture delimited by underscores. This is the default behavior and backwards compatible with deb-pkg-tools 6.0 and older.

If *`PARSE_STRICT`* is `False` then *`parse_filename()`* will fall back to reading the package name, version and architecture from the metadata contained in the `*.deb` archive.

The environment variable `$DPT_PARSE_STRICT` can be used to control the value of this variable (see `coerce_boolean()` for acceptable values).

deb_pkg_tools.package.**ROOT_USER = 'root'**
The name of the system user that is used by *`build_package()`* when it normalizes file ownership using chown (controlled by *`ALLOW_CHOWN`*).

The environment variable `$DPT_ROOT_USER` can be used to control the value of this variable.

deb_pkg_tools.package.**ROOT_GROUP = 'root'**
The name of the system group that is used by *`build_package()`* when it normalizes file ownership using chown (controlled by *`ALLOW_CHOWN`*).

The environment variable `$DPT_ROOT_GROUP` can be used to control the value of this variable.

deb_pkg_tools.package.**parse_filename**(*filename*, *cache=None*)
Parse the filename of a Debian binary package archive.

Parameters

- **filename** – The pathname of a Debian binary package archive (a string).
- **cache** – The *`PackageCache`* to use when *`PARSE_STRICT`* is `False` (defaults to `None`).

Returns A *`PackageFile`* object.

Raises `ValueError` in the following circumstances:

- The filename extension doesn't match any of the known *`BINARY_PACKAGE_ARCHIVE_EXTENSIONS`*.
- The filename doesn't have three underscore separated components (and *`PARSE_STRICT`* is `True`).

This function parses the filename of a Debian binary package archive into three fields: the name of the package, its version and its architecture. See also *`determine_package_archive()`*.

Here's an example:

```
>>> from deb_pkg_tools.package import parse_filename
>>> components = parse_filename('/var/cache/apt/archives/python2.7_2.7.3-0ubuntu3.
↪4_amd64.deb')
>>> print(repr(components))
PackageFile(name='python2.7',
            version='2.7.3-0ubuntu3.4',
```

(continues on next page)

(continued from previous page)

```
            architecture='amd64',
            filename='/var/cache/apt/archives/python2.7_2.7.3-0ubuntu3.4_amd64.deb
↪')
```

class deb_pkg_tools.package.**PackageFile**
A named tuple with the result of *parse_filename()*.

The function *parse_filename()* reports the fields of a package archive's filename as a *PackageFile* object (a named tuple). Here are the fields supported by these named tuples:

name
The name of the package (a string).

version
The version of the package (a *Version* object).

architecture
The architecture of the package (a string).

filename
The absolute pathname of the package archive (a string).

The values of the *directory*, *other_versions* and *newer_versions* properties are generated on demand.

PackageFile objects support sorting according to Debian's package version comparison algorithm as implemented in `dpkg --compare-versions`.

directory
The absolute pathname of the directory containing the package archive (a string).

other_versions
A list of *PackageFile* objects with other versions of the same package in the same directory.

newer_versions
A list of *PackageFile* objects with newer versions of the same package in the same directory.

deb_pkg_tools.package.**find_package_archives**(*directory*, *cache=None*)
Find the Debian package archive(s) in the given directory.

Parameters

- **directory** – The pathname of a directory (a string).
- **cache** – The *PackageCache* that *parse_filename()* should use when *PARSE_STRICT* is `False` (defaults to `None`).

Returns A list of *PackageFile* objects.

deb_pkg_tools.package.**collect_related_packages**(*filename*, *strict=None*, *cache=None*, *interactive=None*)
Collect the package archive(s) related to the given package archive.

Parameters

- **filename** – The filename of an existing `*.deb` archive (a string).
- **cache** – The *PackageCache* to use (defaults to `None`).
- **interactive** – `True` to draw an interactive spinner on the terminal (see `Spinner`), `False` to skip the interactive spinner or `None` to detect whether we're connected to an interactive terminal.

Returns A list of *PackageFile* objects.

This works by parsing and resolving the dependencies of the given package to filenames of package archives, then parsing and resolving the dependencies of those package archives, etc. until no more relationships can be resolved to existing package archives.

Known limitations / sharp edges of this function:

- Only *Depends* and *Pre-Depends* relationships are processed, *Provides* is ignored. I'm not yet sure whether it makes sense to add support for *Conflicts*, *Provides* and *Replaces* (and how to implement it).
- Unsatisfied relationships don't trigger a warning or error because this function doesn't know in what context a package can be installed (e.g. which additional repositories a given apt client has access to).
- Please thoroughly test this functionality before you start to rely on it. What this function tries to do is a complex operation to do correctly (given the limited information this function has to work with) and the implementation is far from perfect. Bugs have been found and fixed in this code and more bugs will undoubtedly be discovered. You've been warned :-).
- This function can be rather slow on large package repositories and dependency sets due to the incremental nature of the related package collection. It's a known issue / limitation.

This function is used to implement the `deb-pkg-tools --collect` command:

```
$ deb-pkg-tools -c /tmp python-deb-pkg-tools_1.13-1_all.deb
2014-05-18 08:33:42 deb_pkg_tools.package INFO Collecting packages related to ~/
↪python-deb-pkg-tools_1.13-1_all.deb ..
2014-05-18 08:33:42 deb_pkg_tools.package INFO Scanning ~/python-deb-pkg-tools_1.
↪13-1_all.deb ..
2014-05-18 08:33:42 deb_pkg_tools.package INFO Scanning ~/python-coloredlogs_0.4.
↪8-1_all.deb ..
2014-05-18 08:33:42 deb_pkg_tools.package INFO Scanning ~/python-chardet_2.2.1-1_
↪all.deb ..
2014-05-18 08:33:42 deb_pkg_tools.package INFO Scanning ~/python-humanfriendly_1.
↪7.1-1_all.deb ..
2014-05-18 08:33:42 deb_pkg_tools.package INFO Scanning ~/python-debian_0.1.21-1_
↪all.deb ..
Found 5 package archives:
 - ~/python-chardet_2.2.1-1_all.deb
 - ~/python-coloredlogs_0.4.8-1_all.deb
 - ~/python-deb-pkg-tools_1.13-1_all.deb
 - ~/python-humanfriendly_1.7.1-1_all.deb
 - ~/python-debian_0.1.21-1_all.deb
Copy 5 package archives to /tmp? [Y/n] y
2014-05-18 08:33:44 deb_pkg_tools.cli INFO Done! Copied 5 package archives to /
↪tmp.
```

deb_pkg_tools.package.**collect_related_packages_helper**(*candidate_archives*, *given_archive*, *cache*, *interactive*)

Internal helper for package collection to enable simple conflict resolution.

deb_pkg_tools.package.**match_relationships**(*package_archive*, *relationship_sets*)

Internal helper for package collection to validate that all relationships are satisfied.

This function enables *collect_related_packages_helper()* to validate that all relationships are satisfied while the set of related package archives is being collected and again afterwards to make sure that no previously drawn conclusions were invalidated by additionally collected package archives.

exception deb_pkg_tools.package.**CollectedPackagesConflict**(*conflicts*)

Exception raised by *collect_related_packages_helper()*.

__init__(*conflicts*)
Construct a *CollectedPackagesConflict* exception.

Parameters **conflicts** – A list of conflicting *PackageFile* objects.

deb_pkg_tools.package.**find_latest_version**(*packages*, *cache=None*)
Find the package archive with the highest version number.

Parameters

- **packages** – A list of filenames (strings) and/or *PackageFile* objects.
- **cache** – The *PackageCache* that *parse_filename()* should use when *PARSE_STRICT* is False (defaults to None).

Returns The *PackageFile* with the highest version number.

Raises ValueError when not all of the given package archives share the same package name.

This function uses *Version* objects for version comparison.

deb_pkg_tools.package.**group_by_latest_versions**(*packages*, *cache=None*)
Group package archives by name of package and find latest version of each.

Parameters

- **packages** – A list of filenames (strings) and/or *PackageFile* objects.
- **cache** – The *PackageCache* that *parse_filename()* should use when *PARSE_STRICT* is False (defaults to None).

Returns A dictionary with package names as keys and *PackageFile* objects as values.

deb_pkg_tools.package.**inspect_package**(*archive*, *cache=None*)
Get the metadata and contents from a *.deb archive.

Parameters

- **archive** – The pathname of an existing *.deb archive.
- **cache** – The *PackageCache* to use (defaults to None).

Returns

A tuple with two dictionaries:

1. The result of *inspect_package_fields()*.
2. The result of *inspect_package_contents()*.

deb_pkg_tools.package.**inspect_package_fields**(*archive*, *cache=None*)
Get the fields (metadata) from a *.deb archive.

Parameters

- **archive** – The pathname of an existing *.deb archive.
- **cache** – The *PackageCache* to use (defaults to None).

Returns A dictionary with control file fields (the result of *parse_control_fields()*).

Here's an example:

```
>>> from deb_pkg_tools.package import inspect_package_fields
>>> print(repr(inspect_package_fields('python3.4-minimal_3.4.0-1+precise1_amd64.
↪deb')))
{'Architecture': u'amd64',
```

(continues on next page)

(continued from previous page)

```
 'Conflicts': RelationshipSet(VersionedRelationship(name=u'binfmt-support',␣
↪operator=u'<<', version=u'1.1.2')),
 'Depends': RelationshipSet(VersionedRelationship(name=u'libpython3.4-minimal',␣
↪operator=u'=', version=u'3.4.0-1+precise1'),
                            VersionedRelationship(name=u'libexpat1', operator=u'>=
↪', version=u'1.95.8'),
                            VersionedRelationship(name=u'libgcc1', operator=u'>=',
↪ version=u'1:4.1.1'),
                            VersionedRelationship(name=u'zlib1g', operator=u'>=',␣
↪version=u'1:1.2.0')),
 'Description': u'Minimal subset of the Python language (version 3.4)\n This␣
↪package contains the interpreter and some essential modules.  It can\n be used␣
↪in the boot process for some basic tasks.\n See /usr/share/doc/python3.4-
↪minimal/README.Debian for a list of the modules\n contained in this package.',
 'Installed-Size': 3586,
 'Maintainer': u'Felix Krull <f_krull@gmx.de>',
 'Multi-Arch': u'allowed',
 'Original-Maintainer': u'Matthias Klose <doko@debian.org>',
 'Package': u'python3.4-minimal',
 'Pre-Depends': RelationshipSet(VersionedRelationship(name=u'libc6', operator=u'>=
↪', version=u'2.15')),
 'Priority': u'optional',
 'Recommends': u'python3.4',
 'Section': u'python',
 'Source': u'python3.4',
 'Suggests': RelationshipSet(Relationship(name=u'binfmt-support')),
 'Version': u'3.4.0-1+precise1'}
```

deb_pkg_tools.package.**inspect_package_contents**(*archive*, *cache=None*)

Get the contents from a `*.deb` archive.

Parameters

- **archive** – The pathname of an existing `*.deb` archive.
- **cache** – The *PackageCache* to use (defaults to None).

Returns A dictionary with the directories and files contained in the package. The dictionary keys are the absolute pathnames and the dictionary values are *ArchiveEntry* objects (see the example below).

An example:

```
>>> from deb_pkg_tools.package import inspect_package_contents
>>> print(repr(inspect_package_contents('python3.4-minimal_3.4.0-1+precise1_amd64.
↪deb')))
{u'/': ArchiveEntry(permissions=u'drwxr-xr-x', owner=u'root', group=u'root',␣
↪size=0, modified=u'2014-03-20 23:54', target=u''),
 u'/usr/': ArchiveEntry(permissions=u'drwxr-xr-x', owner=u'root', group=u'root',␣
↪size=0, modified=u'2014-03-20 23:52', target=u''),
 u'/usr/bin/': ArchiveEntry(permissions=u'drwxr-xr-x', owner=u'root', group=u'root
↪', size=0, modified=u'2014-03-20 23:54', target=u''),
 u'/usr/bin/python3.4': ArchiveEntry(permissions=u'-rwxr-xr-x', owner=u'root',␣
↪group=u'root', size=3536680, modified=u'2014-03-20 23:54', target=u''),
 u'/usr/bin/python3.4m': ArchiveEntry(permissions=u'hrwxr-xr-x', owner=u'root',␣
↪group=u'root', size=0, modified=u'2014-03-20 23:54', target=u'/usr/bin/python3.4
↪'),
 u'/usr/share/': ArchiveEntry(permissions=u'drwxr-xr-x', owner=u'root', group=u
↪'root', size=0, modified=u'2014-03-20 23:53', target=u''),
```

(continues on next page)

(continued from previous page)

```
 u'/usr/share/binfmts/': ArchiveEntry(permissions=u'drwxr-xr-x', owner=u'root',␣
↪group=u'root', size=0, modified=u'2014-03-20 23:53', target=u''),
 u'/usr/share/binfmts/python3.4': ArchiveEntry(permissions=u'-rw-r--r--', owner=u
↪'root', group=u'root', size=72, modified=u'2014-03-20 23:53', target=u''),
 u'/usr/share/doc/': ArchiveEntry(permissions=u'drwxr-xr-x', owner=u'root',␣
↪group=u'root', size=0, modified=u'2014-03-20 23:53', target=u''),
 u'/usr/share/doc/python3.4-minimal/': ArchiveEntry(permissions=u'drwxr-xr-x',␣
↪owner=u'root', group=u'root', size=0, modified=u'2014-03-20 23:54', target=u''),
 u'/usr/share/doc/python3.4-minimal/README.Debian': ArchiveEntry(permissions=u'-
↪rw-r--r--', owner=u'root', group=u'root', size=3779, modified=u'2014-03-20 23:52
↪', target=u''),
 u'/usr/share/doc/python3.4-minimal/changelog.Debian.gz':␣
↪ArchiveEntry(permissions=u'-rw-r--r--', owner=u'root', group=u'root',␣
↪size=28528, modified=u'2014-03-20 22:32', target=u''),
 u'/usr/share/doc/python3.4-minimal/copyright': ArchiveEntry(permissions=u'-rw-r--
↪r--', owner=u'root', group=u'root', size=51835, modified=u'2014-03-20 20:37',␣
↪target=u''),
 u'/usr/share/man/': ArchiveEntry(permissions=u'drwxr-xr-x', owner=u'root',␣
↪group=u'root', size=0, modified=u'2014-03-20 23:52', target=u''),
 u'/usr/share/man/man1/': ArchiveEntry(permissions=u'drwxr-xr-x', owner=u'root',␣
↪group=u'root', size=0, modified=u'2014-03-20 23:54', target=u''),
 u'/usr/share/man/man1/python3.4.1.gz': ArchiveEntry(permissions=u'-rw-r--r--',␣
↪owner=u'root', group=u'root', size=5340, modified=u'2014-03-20 23:30', target=u'
↪'),
 u'/usr/share/man/man1/python3.4m.1.gz': ArchiveEntry(permissions=u'lrwxrwxrwx',␣
↪owner=u'root', group=u'root', size=0, modified=u'2014-03-20 23:54', target=u
↪'python3.4.1.gz')}
```

class deb_pkg_tools.package.**ArchiveEntry**

A named tuple with the result of *inspect_package()*.

The function *inspect_package()* reports the contents of package archives as a dictionary containing named tuples. Here are the fields supported by those named tuples:

permissions
: The entry type and permission bits just like `ls -l` prints them (a string like *drwxr-xr-x*).

owner
: The username of the owner of the entry (a string).

group
: The group name of group owning the entry (a string).

size
: The size of the entry in bytes (an integer).

modified
: A string like `2013-09-26 22:28`.

target
: If the entry represents a symbolic link this field gives the pathname of the target of the symbolic link. Defaults to an empty string.

device_type
: If the entry represents a device file this field gives the device type major and minor numbers as a tuple of two integers. Defaults to a tuple with two zeros.

Note: This defaults to a tuple with two zeros so that *ArchiveEntry* tuples can be reliably sorted just

like regular tuples (i.e. without getting `TypeError` exceptions due to comparisons between incompatible value types).

---

deb_pkg_tools.package.**build_package**(*directory*, *repository=None*, *check_package=True*, *copy_files=True*, ***options*)

Create a Debian package using the `dpkg-deb --build` command.

Parameters

- **directory** – The pathname of a directory tree suitable for packaging with `dpkg-deb --build`.
- **repository** – The pathname of the directory where the generated `*.deb` archive should be stored.

  By default a temporary directory is created to store the generated archive, in this case the caller is responsible for cleaning up the directory.

  Before deb-pkg-tools 2.0 this defaulted to the system wide temporary directory which could result in corrupted archives during concurrent builds.
- **check_package** – If `True` (the default) Lintian is run to check the resulting package archive for possible issues.
- **copy_files** – If `True` (the default) the package's files are copied to a temporary directory before being modified. You can set this to `False` if you're already working on a copy and don't want yet another copy to be made.
- **update_conffiles** – If `True` (the default) files in `/etc` will be added to `DEBIAN/conffiles` automatically using *update_conffiles()*, otherwise it is up to the caller whether to do this or not.
- **strip_object_files** – If `True` (not the default) then *strip_object_files()* will be used.
- **find_system_dependencies** – If `True` (not the default) then *find_system_dependencies()* will be used.

Returns The pathname of the generated `*.deb` archive.

Raises `executor.ExternalCommandFailed` if any of the external commands invoked by this function fail.

The `dpkg-deb --build` command requires a certain directory tree layout and specific files; for more information about this topic please refer to the Debian Binary Package Building HOWTO. The *build_package()* function performs the following steps to build a package:

1. Copies the files in the source directory to a temporary build directory.
2. Updates the Installed-Size field in the `DEBIAN/control` file based on the size of the given directory (using *update_installed_size()*).
3. Sets the owner and group of all files to `root` because this is the only user account guaranteed to always be available. This uses the fakeroot command so you don't actually need `root` access to use *build_package()*.
4. Runs the command `fakeroot dpkg-deb --build` to generate a Debian package from the files in the build directory.
5. Runs Lintian to check the resulting package archive for possible issues. The result of Lintian is purely informational: If 'errors' are reported and Lintian exits with a nonzero status code, this is ignored by *build_package()*.

deb_pkg_tools.package.**determine_package_archive**(*directory*)
Determine the name of a package archive before building it.

Parameters **source_directory** – The pathname of a directory tree suitable for packaging with `dpkg-deb --build`.

Returns The filename of the `*.deb` archive to be built.

This function determines the name of the `*.deb` package archive that will be generated from a directory tree suitable for packaging with `dpkg-deb --build`. See also *parse_filename()*.

deb_pkg_tools.package.**copy_package_files**(*from_directory*, *to_directory*, *hard_links=True*)
Copy package files to a temporary directory, using hard links when possible.

Parameters

- **from_directory** – The pathname of a directory tree suitable for packaging with `dpkg-deb --build`.
- **to_directory** – The pathname of a temporary build directory.
- **hard_links** – Use hard links to speed up copying when possible.

This function copies a directory tree suitable for packaging with `dpkg-deb --build` to a temporary build directory so that individual files can be replaced without changing the original directory tree. If the build directory is on the same file system as the source directory, hard links are used to speed up the copy. This function is used by *build_package()*.

deb_pkg_tools.package.**clean_package_tree**(*directory*, *remove_dirs=('.bzr', '.git', '.hg', '.svn', '__pycache__')*, *remove_files=('*.pyc', '*.pyo', '*~', '.*.s??', '.DS_Store', '.DS_Store.gz', '._*', '.bzrignore', '.gitignore', '.hg_archival.txt', '.hgignore', '.hgtags', '.s??')*)
Clean up files that should not be included in a Debian package from the given directory.

Parameters

- **directory** – The pathname of the directory to clean (a string).
- **remove_dirs** – An iterable with filename patterns of directories that should not be included in the package. Defaults to *DIRECTORIES_TO_REMOVE*.
- **remove_files** – An iterable with filename patterns of files that should not be included in the package. Defaults to *FILES_TO_REMOVE*.

Uses the `fnmatch` module for directory and filename matching. Matching is done on the base name of each directory and file. This function assumes it is safe to unlink files from the given directory (which it should be when *copy_package_files()* was previously called, e.g. by *build_package()*).

deb_pkg_tools.package.**strip_object_files**(*object_files*)
Use strip to make object files smaller.

Parameters **object_files** – An iterable of strings with filenames of object files.

This function runs `strip --strip-unneeded` on each of the given object files to make them as small as possible. To find the object files you can use *find_object_files()*.

If the strip program is not installed a *debug* message is logged but no exceptions are raised. When the strip program fails a *warning* message is logged but again, no exceptions are raised.

One reason not to propagate these error conditions as exceptions is that *find_object_files()* will match files with binary contents that have their executable bit set, regardless of whether those files are actually valid object files.

deb_pkg_tools.package.**find_system_dependencies**(*object_files*)

Use dpkg-shlibdeps to find dependencies on system packages.

Parameters **object_files** – An iterable of strings with filenames of object files.

Returns A list of strings in the format of the entries on the `Depends:` line of a binary package control file.

This function uses the dpkg-shlibdeps program to find dependencies on system packages by analyzing the given object files (binary executables and/or `*.so` files). To find the object files you can use *find_object_files()*.

Here's an example to make things a bit more concrete:

```
>>> find_system_dependencies(['/usr/bin/ssh'])
['libc6 (>= 2.17)',
 'libgssapi-krb5-2 (>= 1.12.1+dfsg-2)',
 'libselinux1 (>= 1.32)',
 'libssl1.0.0 (>= 1.0.1)',
 'zlib1g (>= 1:1.1.4)']
```

Very advanced magic! :-)

deb_pkg_tools.package.**find_object_files**(*directory*)

Find binary executables and `*.so` files.

Parameters **directory** – The pathname of the directory to search (a string).

Returns A list of filenames of object files (strings).

This function is used by *build_package()* to find files to process with *find_system_dependencies()* and *strip_object_files()*. It works by inspecting all of the files in the given *directory*:

- If the filename matches `*.so` it is considered an object file.
- If the file is marked executable and it contains binary data it is also considered an object file, unless the filename matches one of the patterns in *OBJECT_FILE_EXCLUDES*.

deb_pkg_tools.package.**is_binary_file**(*filename*)

Check whether a file appears to contain binary data.

Parameters **filename** – The filename of the file to check (a string).

Returns `True` if the file appears to contain binary data, `False` otherwise.

deb_pkg_tools.package.**update_conffiles**(*directory*)

Make sure the `DEBIAN/conffiles` file is up to date.

Parameters **directory** – The pathname of a directory tree suitable for packaging with `dpkg-deb --build`.

Given a directory tree suitable for packaging with `dpkg-deb --build` this function updates the entries in the `DEBIAN/conffiles` file. This function is used by *build_package()*.

In deb-pkg-tools release 8.4 support for excludes was added: If an entry in the `DEBIAN/conffiles` starts with an exclamation mark (optionally followed by whitespace) that entry will be omitted from the final file.

deb_pkg_tools.package.**update_installed_size**(*directory*)

Make sure the `Installed-Size` field in `DEBIAN/control` is up to date.

Parameters **directory** – The pathname of a directory tree suitable for packaging with `dpkg-deb --build`.

Given a directory tree suitable for packaging with `dpkg-deb --build` this function updates the Installed-Size field in the `DEBIAN/control` file. This function is used by *build_package()*.

2.1.10 deb_pkg_tools.repo

Create, update and activate trivial Debian package repositories.

The functions in the *deb_pkg_tools.repo* module make it possible to transform a directory of `*.deb` archives into a (temporary) Debian package repository:

- *update_repository()* creates/updates a trivial repository
- *activate_repository()* enables `apt-get` to install packages from the trivial repository
- *deactivate_repository()* cleans up after *activate_repository()*

All of the functions in this module can raise `executor.ExternalCommandFailed`.

You can configure the GPG key(s) used by this module through a configuration file, please refer to the documentation of *select_gpg_key()*.

deb_pkg_tools.repo.**ALLOW_SUDO = True**
`True` to enable the use of sudo during operations that normally require elevated privileges (the default), `False` to disable the use of sudo. This option is provided for power users to disable the use of sudo because it may not be available in all build environments. The environment variable `$DPT_SUDO` can be used to control the value of this variable (see `coerce_boolean()` for acceptable values).

deb_pkg_tools.repo.**scan_packages**(*repository*, *packages_file=None*, *cache=None*)
A reimplementation of the `dpkg-scanpackages -m` command in Python.

Updates a `Packages` file based on the Debian package archive(s) found in the given directory. Uses *PackageCache* to (optionally) speed up the process significantly by caching package metadata and hashes on disk. This explains why this function can be much faster than the dpkg-scanpackages program.

Parameters

- **repository** – The pathname of a directory containing Debian package archives (a string).
- **packages_file** – The pathname of the `Packages` file to update (a string). Defaults to the `Packages` file in the given directory.
- **cache** – The *PackageCache* to use (defaults to `None`).

deb_pkg_tools.repo.**get_packages_entry**(*pathname*, *cache=None*)
Get a dictionary with the control fields required in a `Packages` file.

Parameters

- **pathname** – The pathname of the package archive (a string).
- **cache** – The *PackageCache* to use (defaults to `None`).

Returns A dictionary with control fields (see below).

Used by *scan_packages()* to generate `Packages` files. The format of `Packages` files (part of the Debian binary package repository format) is fairly simple:

- All of the fields extracted from a package archive's control file using *inspect_package_fields()* are listed (you have to get these fields yourself and combine the dictionaries returned by *inspect_package_fields()* and *get_packages_entry()*);
- The field `Filename` contains the filename of the package archive relative to the `Packages` file (which is in the same directory in our case, because *update_repository()* generates trivial repositories);

- The field `Size` contains the size of the package archive in bytes;
- The following fields contain package archive checksums:

  MD5sum Calculated using the `md5()` constructor of the `hashlib` module.

  SHA1 Calculated using the `sha1()` constructor of the `hashlib` module.

  SHA256 Calculated using the `sha256()` constructor of the `hashlib` module.

The three checksums are calculated simultaneously by reading the package archive once, in blocks of a kilobyte. This is probably why this function seems to be faster than `dpkg-scanpackages -m` (even when used without caching).

`deb_pkg_tools.repo.`**`update_repository`**(*directory*, *release_fields={}*, *gpg_key=None*, *cache=None*)

Create or update a trivial repository.

Parameters

- **directory** – The pathname of a directory with `*.deb` packages.
- **release_fields** – An optional dictionary with fields to set inside the `Release` file.
- **gpg_key** – The *GPGKey* object used to sign the repository. Defaults to the result of *select_gpg_key()*.
- **cache** – The *PackageCache* to use (defaults to `None`).

Raises *ResourceLockedException* when the given repository directory is being updated by another process.

This function is based on the Debian programs dpkg-scanpackages and apt-ftparchive and also uses gpg and gzip. The following files are generated:

File-name	Description
`Packages`	Provides the metadata of all `*.deb` packages in the trivial repository as a single text file. Generated using *scan_packages()* (as a faster alternative to dpkg-scanpackages).
`Packages.gz`	A compressed version of the package metadata generated using gzip.
`Release`	Metadata about the release and hashes of the `Packages` and `Packages.gz` files. Generated using apt-ftparchive.
`Release.gpg`	An ASCII-armored detached GPG signature of the `Release` file. Generated using `gpg --armor --sign --detach-sign`.
`InRelease`	The contents of the `Release` file and its GPG signature combined into a single human readable file. Generated using `gpg --armor --sign --clearsign`.

For more details about the `Release.gpg` and `InRelease` files please refer to the Debian wiki's section on secure-apt.

`deb_pkg_tools.repo.`**`activate_repository`**(*directory*, *gpg_key=None*)

Activate a local trivial repository.

Parameters

- **directory** – The pathname of a directory with `*.deb` packages.
- **gpg_key** – The *GPGKey* object used to sign the repository. Defaults to the result of *select_gpg_key()*.

This function sets everything up so that a trivial Debian package repository can be used to install packages without a webserver. This uses the `file://` URL scheme to point apt-get to a directory on the local file system.

> **Warning:** This function requires `root` privileges to:
>
> 1. create the directory `/etc/apt/sources.list.d`,
> 2. create a `*.list` file in `/etc/apt/sources.list.d` and
> 3. run `apt-get update`.
>
> This function will use sudo to gain `root` privileges when it's not already running as `root`.

See also:

ALLOW_SUDO

deb_pkg_tools.repo.**deactivate_repository**(*directory*)

Deactivate a local repository that was previously activated using *activate_repository()*.

Parameters **directory** – The pathname of a directory with `*.deb` packages.

> **Warning:** This function requires `root` privileges to:
>
> 1. delete a `*.list` file in `/etc/apt/sources.list.d` and
> 2. run `apt-get update`.
>
> This function will use sudo to gain `root` privileges when it's not already running as `root`.

See also:

ALLOW_SUDO

deb_pkg_tools.repo.**with_repository**(*directory*, **command*, ***kw*)

Execute an external command while a repository is activated.

Parameters

- **directory** – The pathname of a directory containing `*.deb` archives (a string).
- **command** – The command to execute (a tuple of strings, passed verbatim to `executor.execute()`).
- **cache** – The *PackageCache* to use (defaults to `None`).

Raises `executor.ExternalCommandFailed` if any external commands fail.

This function create or updates a trivial package repository, activates the repository, runs an external command (usually `apt-get install`) and finally deactivates the repository again. Also deactivates the repository when the external command fails and `executor.ExternalCommandFailed` is raised.

See also:

ALLOW_SUDO

deb_pkg_tools.repo.**apt_supports_trusted_option**()

Figure out whether apt supports the `[trusted=yes]` option.

Returns `True` if the option is supported, `False` if it is not.

Since apt version 0.8.16~exp3 the option `[trusted=yes]` can be used in a `sources.list` file to disable GPG key checking (see Debian bug #596498). This version of apt is included with Ubuntu 12.04 and later, but deb-pkg-tools also has to support older versions of apt. The *apt_supports_trusted_option()* function checks if the installed version of apt supports the `[trusted=yes]` option, so that deb-pkg-tools can use it when possible.

deb_pkg_tools.repo.**select_gpg_key**(*directory*)

Select a suitable GPG key for repository signing.

Parameters **directory** – The pathname of the directory that contains the package repository to sign (a string).

Returns A *GPGKey* object or `None`.

Used by *update_repository()* and *activate_repository()* to select the GPG key for repository signing based on a configuration file.

Configuration file locations:

The following locations are checked for a configuration file:

1. `~/.deb-pkg-tools/repos.ini`
2. `/etc/deb-pkg-tools/repos.ini`

If both files exist only the first one is used.

Configuration file contents:

The configuration files are in the `*.ini` file format (refer to the `ConfigParser` module for details). Each section in the configuration file defines a signing key.

The `directory` option controls to which directory or directories a signing key applies. The value of this option is the pathname of a directory and supports pattern matching using `?` and `*` (see the `fnmatch` module for details).

The default signing key:

If a section does not define a `directory` option then that section is used as the default signing key for directories that are not otherwise matched (by a `directory` option).

Compatibility with GnuPG >= 2.1:

GnuPG 2.1 compatibility was implemented in deb-pkg-tools release 5.0 which changes how users are expected to select an isolated GPG key pair:

- Before deb-pkg-tools 5.0 only GnuPG < 2.1 was supported and the configuration used the `public-key-file` and `secret-key-file` options to configure the pathnames of the public key file and the secret key file:

```
[old-example]
public-key-file = ~/.deb-pkg-tools/default-signing-key.pub
secret-key-file = ~/.deb-pkg-tools/default-signing-key.sec
```

- In deb-pkg-tools 5.0 support for GnuPG >= 2.1 was added which means the public key and secret key files are no longer configured separately, instead a `key-store` option is used to point to a directory in the format of `~/.gnupg` containing the key pair:

```
[new-example]
key-store = ~/.deb-pkg-tools/default-signing-key/
```

Additionally a `key-id` option was added to make it possible to select a specific key pair from a GnuPG profile directory.

Staying backwards compatible:

By specifying all three of the `public-key-file`, `secret-key-file` and `key-store` options it is possible to achieve compatibility with all supported GnuPG versions:

- When GnuPG >= 2.1 is installed the `key-store` option will be used.
- When GnuPG < 2.1 is installed the `public-key-file` and `secret-key-file` options will be used.

In this case the caller is responsible for making sure that a suitable key pair is available in both locations (compatible with the appropriate version of GnuPG).

Default behavior:

If no GPG keys are configured but apt requires local repositories to be signed (see *apt_supports_trusted_option()*) then this function falls back to selecting an automatically generated signing key. The generated key pair is stored in the directory `~/.deb-pkg-tools`.

`deb_pkg_tools.repo.`**`load_config`**(*repository*)
Load repository configuration from a `repos.ini` file.

2.1.11 deb_pkg_tools.utils

Utility functions.

The functions in the *deb_pkg_tools.utils* module are not directly related to Debian packages/repositories, however they are used by the other modules in the *deb-pkg-tools* package.

`deb_pkg_tools.utils.`**`compact`**(*text*, **args*, ***kw*)
Alias for backwards compatibility.

`deb_pkg_tools.utils.`**`sha1`**(*text*)
Calculate the SHA1 fingerprint of text.

Parameters **`text`** – The text to fingerprint (a string).

Returns The fingerprint of the text (a string).

`deb_pkg_tools.utils.`**`makedirs`**(*directory*)
Create a directory and any missing parent directories.

It is not an error if the directory already exists.

Parameters **`directory`** – The pathname of a directory (a string).

Returns `True` if the directory was created, `False` if it already exists.

`deb_pkg_tools.utils.`**`optimize_order`**(*package_archives*)
Shuffle a list of package archives in random order.

Usually when scanning a large group of package archives, it really doesn't matter in which order we scan them. However the progress reported using `Spinner` can be more accurate when we shuffle the order. Why would that happen? When the following conditions are met:

1. The package repository contains multiple versions of the same packages;
2. The package repository contains both small and (very) big packages.

If you scan the package archives in usual sorting order you will first hit a batch of multiple versions of the same small package which can be scanned very quickly (the progress counter will jump). Then you'll hit a batch of multiple versions of the same big package and scanning becomes much slower (the progress counter will hang). Shuffling mostly avoids this effect.

deb_pkg_tools.utils.**find_debian_architecture**()
Find the Debian architecture of the current environment.

Uses `os.uname()` to determine the current machine architecture (the fifth value returned by `os.uname()`) and translates it into one of the machine architecture labels used in the Debian packaging system:

Machine architecture	Debian architecture
i686	i386
x86_64	amd64
armv6l	armhf

When the machine architecture is not listed above, this function falls back to the external command `dpkg-architecture -qDEB_BUILD_ARCH` (provided by the `dpkg-dev` package). This command is not used by default because:

1. deb-pkg-tools doesn't have a strict dependency on `dpkg-dev`.
2. The dpkg-architecture program enables callers to set the current architecture and the exact semantics of this are unclear to me at the time of writing (it can't automagically provide a cross compilation environment, so what exactly does it do?).

Returns The Debian architecture (a string like `i386`, `amd64`, `armhf`, etc).

Raises `ExternalCommandFailed` when the dpkg-architecture program is not available or reports an error.

deb_pkg_tools.utils.**find_installed_version**(*package_name*)
Find the installed version of a Debian system package.

Parameters **package_name** – The name of the package (a string).

Returns The installed version of the package (a string) or `None` if the version can't be found.

This function uses the `dpkg-query --show --showformat='${Version}' ...` command (see the dpkg-query documentation for details).

class deb_pkg_tools.utils.**atomic_lock**(*pathname*, *wait=True*)
Context manager for atomic locking of files and directories.

This context manager exploits the fact that `os.mkdir()` on UNIX is an atomic operation, which means it will only work on UNIX.

Intended to be used with Python's `with` statement:

```
with atomic_lock('/var/www/apt-archive/some/repository'):
   # Inside the with block you have exclusive access.
   pass
```

__init__(*pathname*, *wait=True*)
Prepare to atomically lock the given pathname.

Parameters

- **pathname** – The pathname of a file or directory (a string).
- **wait** – Block until the lock can be claimed (a boolean, defaults to `True`).

If `wait=False` and the file or directory cannot be locked, *ResourceLockedException* will be raised when entering the `with` block.

__enter__()
Atomically lock the given pathname.

__exit__(*exc_type=None*, *exc_value=None*, *traceback=None*)
Unlock the previously locked pathname.

exception deb_pkg_tools.utils.**ResourceLockedException**
Raised by *atomic_lock()* when the lock can't be claimed.

2.1.12 deb_pkg_tools.version

Version comparison and sorting according to Debian semantics.

The *deb_pkg_tools.version* module supports version comparison and sorting according to section 5.6.12 of the Debian Policy Manual. The main entry points for users of the Python API are the *compare_versions()* function and the *Version* class.

This module contains two Debian version comparison implementations:

compare_versions_native() This is a pure Python implementation of the Debian version sorting algorithm. It's the default choice of *compare_versions()* for performance reasons.

compare_versions_external() This works by running the external command `dpkg --compare-versions`. It's provided only as an alternative to fall back on should issues come to light with the implementation of *compare_versions_native()*, for more on that please refer to *PREFER_DPKG*.

Note: Deprecated names

The following aliases exist to preserve backwards compatibility, however a `DeprecationWarning` is triggered when they are accessed, because these aliases will be removed in a future release.

deb_pkg_tools.version.**dpkg_comparison_cache**
Alias for *deb_pkg_tools.version.DPKG_COMPARISON_CACHE*.

deb_pkg_tools.version.**compare_versions_with_dpkg**
Alias for *deb_pkg_tools.version.compare_versions_external*.

deb_pkg_tools.version.**compare_versions_with_python_apt**
Alias for *deb_pkg_tools.version.compare_versions_external*.

---

deb_pkg_tools.version.**PREFER_DPKG = False**
`True` to prefer *compare_versions_external()* over *compare_versions_native()*, `False` otherwise (the default is `False`).

The environment variable `$DPT_VERSION_COMPAT` can be used to control the value of this variable (see `coerce_boolean()` for acceptable values).

Note: This option was added in preparation for release 8.0 which replaces python-apt based version comparison with a pure Python implementation that -although tested- definitely has the potential to cause regressions. If regressions do surface this option provides an easy to use "escape hatch" to restore compatibility.

---

deb_pkg_tools.version.**DPKG_COMPARISON_CACHE = {}**
This dictionary is used by *compare_versions_external()* to cache `dpkg --compare-versions` results. Each key in the dictionary is a tuple of three values: (version1, operator, version2). Each value in the dictionary is a boolean (`True` if the comparison succeeded, `False` if it failed).

deb_pkg_tools.version.**NATIVE_COMPARISON_CACHE = {}**
This dictionary is used by *compare_versions_native()* to cache the results of comparisons between version strings. Each key in the dictionary is a tuple of two values: (version1, version2). Each value is one of the following integers:

- -1 means version1 sorts before version2
- 0 means version1 and version2 are equal
- 1 means version1 sorts after version2

This cache is a lot more efficient than *DPKG_COMPARISON_CACHE* because the cache key doesn't contain operators.

deb_pkg_tools.version.**coerce_version**(*value*)
Coerce strings to *Version* objects.

Parameters **value** – The value to coerce (a string or *Version* object).

Returns A *Version* object.

deb_pkg_tools.version.**compare_versions**(*version1*, *operator*, *version2*)
Compare Debian package versions using the best available method.

Parameters

- **version1** – The version on the left side of the comparison (a string).
- **operator** – The operator to use in the comparison (a string).
- **version2** – The version on the right side of the comparison (a string).

Returns `True` if the comparison succeeds, `False` if it fails.

This function prefers to use *compare_versions_native()* but will use *compare_versions_external()* instead when *PREFER_DPKG* is `True`.

deb_pkg_tools.version.**compare_versions_external**(*version1*, *operator*, *version2*)
Compare Debian package versions using the external command `dpkg --compare-versions ...`.

Parameters

- **version1** – The version on the left side of the comparison (a string).
- **operator** – The operator to use in the comparison (a string).
- **version2** – The version on the right side of the comparison (a string).

Returns `True` if the comparison succeeds, `False` if it fails.

See also:

DPKG_COMPARISON_CACHE and *PREFER_DPKG*

deb_pkg_tools.version.**compare_versions_native**(*version1*, *operator*, *version2*)
Compare Debian package versions using a pure Python implementation.

Parameters

- **version1** – The version on the left side of the comparison (a string).
- **operator** – The operator to use in the comparison (a string).
- **version2** – The version on the right side of the comparison (a string).

Returns `True` if the comparison succeeds, `False` if it fails.

See also:

NATIVE_COMPARISON_CACHE and *compare_version_objects()*

class deb_pkg_tools.version.**Version**(*value*)

Rich comparison of Debian package versions as first-class Python objects.

The *Version* class is a subclass of the built in `str` type that implements rich comparison according to the version sorting order defined in the Debian Policy Manual. Use it to sort Debian package versions like this:

```
>>> from deb_pkg_tools.version import Version
>>> unsorted = ['0.1', '0.5', '1.0', '2.0', '3.0', '1:0.4', '2:0.3']
>>> print(sorted(Version(s) for s in unsorted))
['0.1', '0.5', '1.0', '2.0', '3.0', '1:0.4', '2:0.3']
```

This example uses 'epoch' numbers (the numbers before the colons) to demonstrate that this version sorting order is different from regular sorting and 'natural order sorting'.

epoch

The integer value of the epoch number specified by the version string (defaults to zero in case the Debian version number doesn't specify an epoch number).

upstream_version

A string containing the main version number component that encodes the upstream version number.

debian_revision

A string containing the Debian revision suffixed to the version number.

__init__(*value*)

Initialize a *Version* object.

Parameters **value** – A string containing a Debian version number.

__hash__()

Enable adding *Version* objects to sets and using them as dictionary keys.

__eq__(*other*)

Enable equality comparison between *Version* objects.

__ne__(*other*)

Enable non-equality comparison between version objects.

__lt__(*other*)

Enable less-than comparison between version objects.

__le__(*other*)

Enable less-than-or-equal comparison between version objects.

__gt__(*other*)

Enable greater-than comparison between version objects.

__ge__(*other*)

Enable greater-than-or-equal comparison between version objects.

2.1.13 deb_pkg_tools.version.native

Pure Python implementation of Debian version comparison and sorting.

The *deb_pkg_tools.version* module previously integrated with python-apt, however it was pointed out to me in issue #20 that python-apt uses the GPL2 license. Because GPL2 is a viral license it dictates that deb-pkg-tools also needs to be published under GPL2. Because I didn't feel like switching from MIT to GPL I decided to remove

the dependency instead (switching would have cascaded down to several other Python packages I've published and I wasn't comfortable with that).

While working on this pure Python implementation I was initially worried about performance being much worse than using python-apt, so much so that I'd already started researching how to implement a binary "speedup" module. Imagine my surprise when I started running benchmarks and found that my pure Python implementation was (just slightly) faster than python-apt!

deb_pkg_tools.version.native.**compare_strings**(*version1*, *version2*)
Compare two upstream version strings or Debian revision strings.

Parameters

- **version1** – An upstream version string or Debian revision string.
- **version2** – An upstream version string or Debian revision string.

Returns

One of the following integer numbers:

- -1 means version1 sorts before version2
- 0 means version1 and version2 are equal
- 1 means version1 sorts after version2

This function is used by *compare_version_objects()* to perform the comparison of Debian version strings.

deb_pkg_tools.version.native.**compare_version_objects**(*version1*, *version2*)
Compare two *Version* objects.

Parameters

- **version1** – The version on the left side of the comparison (a *Version* object).
- **version2** – The version on the right side of the comparison (a *Version* object).

Returns

One of the following integer numbers:

- -1 means version1 sorts before version2
- 0 means version1 and version2 are equal
- 1 means version1 sorts after version2

This function is used by *compare_versions_native()* to perform the comparison of Debian version strings, after which the operator is interpreted by *compare_versions_native()*.

deb_pkg_tools.version.native.**get_digit_prefix**(*characters*)
Get the digit prefix from a given list of characters.

Parameters **characters** – A list of characters.

Returns An integer number (defaults to zero).

Used by *compare_strings()* as part of the implementation of *compare_versions_native()*.

deb_pkg_tools.version.native.**get_non_digit_prefix**(*characters*)
Get the non-digit prefix from a given list of characters.

Parameters **characters** – A list of characters.

Returns A list of leading non-digit characters (may be empty).

Used by *compare_strings()* as part of the implementation of *compare_versions_native()*.

deb_pkg_tools.version.native.**get_order_mapping**()
Generate a mapping of characters to integers representing sorting order.

Returns A dictionary with string keys and integer values.

Used by *compare_strings()* as part of the implementation of *compare_versions_native()*.

CHAPTER 3

Change log

The change log lists notable changes to the project:

3.1 Changelog

The purpose of this document is to list all of the notable changes to this project. The format was inspired by Keep a Changelog. This project adheres to semantic versioning.

3.1.1 Release 8.4 (2021-03-09)

Enhance *deb_pkg_tools.package.update_conffiles()* with exclude support: If an entry in the `DEBIAN/conffiles` starts with an exclamation mark (optionally followed by whitespace) that entry will be omitted from the final file.

Rationale: In general I like the automatic `DEBIAN/conffiles` updating but I've encountered circumstances in which it is really inconvenient not being able to exclude one or two specific files.

3.1.2 Release 8.3 (2020-05-11)

Minor improvements to the *deb_pkg_tools.deb822* module:

Slightly relax deb822 parsing Leading and trailing comment blocks and empty lines that directly precede or follow a paragraph of control fields are now silently ignored. This is intended to improve compatibility with python-debian.

Improve deb822 parse errors Shortly after I started using deb-pkg-tools 8.0 it became apparent that *deb_pkg_tools.deb822.parse_deb822()* is quite a bit more strict than the previous usage of python-debian. While I don't necessarily consider this a bad thing, it definitely highlighted a weak spot: The error messages didn't include filenames or line numbers. This is now fixed.

3.1.3 Release 8.2 (2020-05-02)

Removed `textwrap.indent()` usage from *deb_pkg_tools.deb822* module because this function isn't available on Python 2.7 which deb-pkg-tools still supports. Also added a regression test.

Note: While I definitely intend to drop Python 2 support in my open source projects at some point, right now is not the time for that just yet.

3.1.4 Release 8.1 (2020-04-25)

- Merged pull request #22 which avoids a `ValueError` exception in the *inspect_package_contents()* function when a device file entry is parsed.
- Enhanced the *inspect_package_contents()* function to properly parse device file type information exposed via the new *ArchiveEntry.device_type* attribute.
- Added a regression test for device file type parsing.

3.1.5 Release 8.0 (2020-04-25)

Dropped GPL2 dependencies The main purpose of this release was to resolve issue #20 by dropping two GPL2 dependencies to avoid having to change the deb-pkg-tools license from MIT to GPL2:

python-apt This dependency was previously used for Debian version comparison. This functionality has now been implemented in pure Python, for more details please refer to the new *deb_pkg_tools.version.native* module.

Note: If this change introduces regressions for you, take a look at the *deb_pkg_tools.version.PREFER_DPKG* variable, it may help as a temporary workaround. Also please report the regression .

---

python-debian This dependency was previously used for Debian binary control file parsing. This functionality has now been implemented in pure Python, for more details please refer to the new *deb_pkg_tools.deb822* module.

Updated Python compatibility Python 3.8 is now officially supported, 3.4 is no longer supported.

Fixed deprecation warnings Fixed humanfriendly 8.0 deprecation warnings and bumped requirements I authored that went through the same process. Also defined the first deprecated aliases in the deb-pkg-tools code base (in the process of implementing the functionality required to drop the GPL2 dependencies).

Quality boost for deb_pkg_tools.control module The *deb_pkg_tools.control* module saw a lot of small changes to make the handling of case insensitivity and byte strings versus Unicode strings more consistent. The most important changes:

- All functions that return dictionaries now return the same type of case insensitive dictionaries (see *Deb822*).
- The complete module now expects and uses Unicode strings internally. Character encoding and decoding is only done when control files are read from and written to disk.

3.1.6 Release 7.0 (2020-02-07)

Code changes:

- Make *update_conffiles()* optional (requested in #19) in the Python API.
- Make *find_object_files()* use a builtin exclude list of filename patterns to ignore.
- Start using `__all__` to control what is exported:
  - This change is backwards incompatible in the sense that until now imports were exposed to the outside world, however for anyone to actually use this would imply not having read the documentation, so this doesn't really bother me.
  - In theory this change could be backwards incompatible in a bad way if I omitted `__all__` entries that should have been exported. I did double check but of course I can't be 100% sure (the `deb_pkg_tools.*` modules currently span almost 6000 lines including whitespace and comments).
  - I decided to bump the major version number because of the potential for import errors caused by the introduction of `__all__`.

Documentation updates:

- Simplified the overview of environment variables in the readme by properly documenting individual options and linking to their documentation entries. Over the years I've picked up the habit of treating my documentation just like my code: Make sure everything is defined in a single place (DRY), as close as possible to the place where it is used. Properly documenting all of the module variables that are based on environment variables and linking to those from the readme frees me from the burden of explaining things in more than one place. This is good because multiple explanations increase the chance of documentation becoming outdated or contradictoring itself, which are definitely problems to be avoided whenever possible.
- Started using `:man:` role to link to Linux manual pages.
- Changed Read the Docs URL (`s/\.org$/.io/g`).

Documented variables:

Module variable	Environment variable
deb_pkg_tools.gpg.FORCE_ENTROPY	$DPT_FORCE_ENTROPY
deb_pkg_tools.package.ALLOW_CHOWN	$DPT_CHOWN_FILES
deb_pkg_tools.package.ALLOW_FAKEROOT_OR_SUDO	$DPT_ALLOW_FAKEROOT_OR_SUDO
deb_pkg_tools.package.ALLOW_HARD_LINKS	$DPT_HARD_LINKS
deb_pkg_tools.package.ALLOW_RESET_SETGID	$DPT_RESET_SETGID
deb_pkg_tools.package.BINARY_PACKAGE_ARCHIVE_EXTENSIONS	
deb_pkg_tools.package.DEPENDENCY_FIELDS	
deb_pkg_tools.package.DIRECTORIES_TO_REMOVE	
deb_pkg_tools.package.FILES_TO_REMOVE	
deb_pkg_tools.package.PARSE_STRICT	$DPT_PARSE_STRICT
deb_pkg_tools.package.ROOT_GROUP	$DPT_ROOT_GROUP
deb_pkg_tools.package.ROOT_USER	$DPT_ROOT_USER
deb_pkg_tools.repo.ALLOW_SUDO	$DPT_SUDO

3.1.7 Release 6.1 (2020-02-05)

Implemented a feature requested from me via private email:

Problem: When filename parsing of `*.deb` archives fails to recognize a package name, version and architecture encoded in the filename (delimited by underscores) then deb-pkg-tools reports an error and aborts:

```
ValueError: Filename doesn't have three underscore separated components!
```

Solution: Setting the environment variable `$DPT_PARSE_STRICT` to `false` changes this behavior so that the required information is extracted from the package metadata instead of reporting an error.

For now the default remains the same (an error is reported) due to backwards compatibility and the principle of least surprise (for those who previously integrated deb-pkg-tools). This will likely change in the future.

Miscellaneous changes:

- Use 'console' highlighting in readme (prompt are now highlighted).
- Added license=MIT to `setup.py` script.
- Bumped copyright to 2020.

3.1.8 Release 6.0 (2019-09-13)

- Enable compatibility with newer python-apt releases:
  - The test suite has been modified to break on Travis CI when python-apt should be available but isn't (when the Python virtual environment is based on a Python interpreter provided by Ubuntu, currently this applies to all build environments except Python 3.7).
  - The idea behind the test suite change is to verify that the conditional import chain in `version.py` always succeeds (on Travis CI, where I control the runtime environment).
  - This was added when after much debugging I finally realized why the new Ubuntu 18.04 build server I'd created was so awfully slow: The conditional import chain had been "silently broken" without me realizing it, except for the fact that using the fall back implementation based on `dpkg --compare-versions` to sort through thousands of version numbers was rather noticeably slow...
- Make python-memcached an optional dependency in response to #13.
- Dropped Python 2.6 compatibility.

3.1.9 Release 5.2 (2018-11-17)

Promote python-debian version constraint into a conditional dependency.

Recently I constrained the version of python-debian to work around a Python 2.6 incompatibility. This same incompatibility is now biting me in the py2deb setup on Travis CI and after fighting that situation for a while I decided it may be better (less convoluted) to fix this in deb-pkg-tools instead (at the source of the problem, so to speak).

3.1.10 Release 5.1.1 (2018-10-26)

Bug fix for logic behind `deb_pkg_tools.GPGKey.existing_files` property: The configured `directory` wasn't being scanned in combination with GnuPG < 2.1 even though the use of `directory` has become the preferred way to configure GnuPG < 2.1 as well as GnuPG >= 2.1 (due to the GnuPG bug mentioned in the release notes of release 5.1).

3.1.11 Release 5.1 (2018-10-26)

Added the `deb_pkg_tools.gpg.GPGKey.identifier` property that uses the `gpg --list-keys --with-colons` command to introspect the key pair and extract a unique identifier:

- When a fingerprint is available in the output this is the preferred value.
- Otherwise the output is searched for a key ID.

If neither of these values is available an exception is raised.

Note: While testing this I noticed that the old style `gpg --no-default-keyring --keyring=... --secret-keyring=...` commands don't support the `--list-keys` command line option. The only workaround for this is to use the `directory` property (which triggers the use of `--homedir`) instead of the `public_key_file` and `secret_key_file` properties. This appears to be due to a bug in older GnuPG releases (see this mailing list thread).

3.1.12 Release 5.0 (2018-10-25)

GnuPG >= 2.1 compatibility for repository signing.

This release became rather more involved than I had hoped it would because of backwards incompatibilities in GnuPG >= 2.1 that necessitated changes in the API that deb-pkg-tools presents to its users:

- The `--secret-keyring` option has been obsoleted and is ignored and the suggested alternative is the use of an ephemeral home directory which changes how a key pair is specified.
- This impacts the API of the `deb_pkg_tools.gpg.GPGKey` class as well as the `repos.ini` support in `deb_pkg_tools.repo.update_repository()`.

The documentation has been updated to explain all of this, refer to the `deb_pkg_tools.gpg` module for details. Detailed overview of changes:

- The `deb_pkg_tools.gpg.GPGKey` class is now based on `property-manager` and no longer uses instance variables, because this made it easier for me to split up the huge `__init__()` method into manageable chunks.

  A side effect is that `__init__()` no longer supports positional arguments which technically speaking is **backwards incompatible** (although I never specifically intended it to be used like that).

- The `deb_pkg_tools.gpg.GPGKey` class now raises an exception when it detects that the use of an isolated key pair is intended but the `directory` option has not been provided even though GnuPG >= 2.1 is being used. While this exception is new, the previous behavior on GnuPG >= 2.1 was anything but sane, so any thoughts about the backwards compatibility of this new exception are a moot point.
- The `deb_pkg_tools.gpg.GPGKey` used to raise `TypeError` when a key pair is explicitly specified but only one of the two expected files exists, in order to avoid overwriting files not "owned" by deb-pkg-tools. An exception is still raised but the type has been changed to `EnvironmentError` because I felt that it was more appropriate. This is technically **backwards incompatible** but I'd be surprised if this affects even a single user...
- The repository activation fall back test (that generates an automatic signing key in order to generate `Release.gpg`) was failing for me on Ubuntu 18.04 and in the process of debugging this I added support for `InRelease` files. In the end this turned out to be irrelevant to the issue at hand, but I saw no harm in keeping the `InRelease` support. This is under the assumption that the presence of an `InRelease` file shouldn't disturb older `apt-get` versions (which seems like a sane assumption to me - it's just a file on a webserver, right?).
- Eventually I found out that the repository activation fall back test was failing due to the key type of the automatic signing key that's generated during the test: As soon as I changed that from DSA to RSA things started working.
- GnuPG profile directory initialization now applies 0700 permissions to avoid noisy warnings from GnuPG.
- Added Python 3.7 to tested and and supported versions.
- Improved `update_repository()` documentation.
- Moved function result caching to `humanfriendly.decorators`.
- I've changed `Depends` to `Recommends` in `stdeb.cfg`, with the following rationale:
  - The deb-pkg-tools package provides a lot of loosely related functionality depending on various external commands. For example building of Debian binary packages requires quite a few programs to be installed.
  - But not every use case of deb-pkg-tools requires all of these external commands, so demanding that they always be installed is rather inflexible.
  - In my specific case this dependency creep blocked me from building lightweight tools on top of deb-pkg-tools, because the dependency chain would pull in a complete build environment. That was more than I bargained for when I wanted to use a few utility functions in deb-pkg-tools .
  - With this change, users are responsible for installing the appropriate packages. But then I estimate that less than one percent of my users are actually affected by this change, because of the low popularity of solutions like stdeb and py2deb .
  - Only the python-apt package remains as a strict dependency instead of a recommended dependency, see 757286fc8ce for the rationale.
- Removed python-apt intersphinx reference (for now).
- Added this changelog to the repository and documentation.

3.1.13 Release 4.5 (2018-02-25)

Improved robustness of `dpkg-shlibdeps` and `strip` integration (followup to release 4.4).

3.1.14 Release 4.4 (2018-02-25)

Integrated support for `dpkg-shlibdeps` (inspired by py2deb).

I first started (ab)using `dpkg-shlibdeps` in the py2deb project and have since missed this functionality in other projects like deb-pkg-tools so have decided to move some stuff around :-).

3.1.15 Release 4.3 (2018-02-25)

- Make mandatory control field validation reusable.
- Include documentation in source distributions.
- Restore Python 2.6 compatibility in test suite.

3.1.16 Release 4.2 (2017-07-10)

Implement cache invalidation (follow up to #12).

3.1.17 Release 4.1 (2017-07-10)

- Merged pull request #11: State purpose of project in readme.
- Improve dependency parsing: Add more `Depends` like fields (fixes #12).
- Start using `humanfriendly.testing` to mark skipped tests.
- Changed Sphinx documentation theme.
- Add Python 3.6 to tested versions.

3.1.18 Release 4.0.2 (2017-02-02)

Bug fix for inheritance of `AlternativeRelationship`. This fixes the following error when hashing relationship objects:

```
AttributeError: 'AlternativeRelationship' object has no attribute 'operator'
```

I'd like to add tests for this but lack the time to do so at this moment, so hopefully I can revisit this later when I have a bit more time .

3.1.19 Release 4.0.1 (2017-02-01)

- Bug fix: Swallow unpickling errors instead of propagating them.

  In general I am very much opposed to Python code that swallows exceptions when it doesn't know how to handle them, because it can inadvertently obscure an issue's root cause and/or exacerbate the issue.

  But caching deserves an exception. Any code that exists solely as an optimization should not raise exceptions caused by the caching logic. This should avoid the following traceback which I just ran into:

```
Traceback (most recent call last):
  File ".../lib/python2.7/site-packages/deb_pkg_tools/cli.py", line 382, in with_
↪repository_wrapper
    with_repository(directory, \*command, cache=cache)
  File ".../lib/python2.7/site-packages/deb_pkg_tools/repo.py", line 366, in with_
↪repository
    cache=kw.get('cache'))
  File ".../lib/python2.7/site-packages/deb_pkg_tools/repo.py", line 228, in␣
↪update_repository
    cache=cache)
  File ".../lib/python2.7/site-packages/deb_pkg_tools/repo.py", line 91, in scan_
↪packages
```

(continues on next page)

(continued from previous page)

```
    fields = dict(inspect_package_fields(archive, cache=cache))
  File ".../lib/python2.7/site-packages/deb_pkg_tools/package.py", line 480, in␣
↪inspect_package_fields
    value = entry.get_value()
  File ".../lib/python2.7/site-packages/deb_pkg_tools/cache.py", line 268, in get_
↪value
    from_fs = pickle.load(handle)
ValueError: unsupported pickle protocol: 3
```

- Added `property-manager` to intersphinx mapping (enabling links in the online documentation).

3.1.20 Release 4.0 (2017-01-31)

- **Added support for parsing of architecture restrictions** (#9).
- Switched `deb_pkg_tools.deps` to use `property-manager` and removed `cached-property` requirement in the process:
  - This change simplified the deb-pkg-tools code base by removing the `deb_pkg_tools.compat.total_ordering` and `deb_pkg_tools.utils.OrderedObject` classes.
  - The introduction of `property-manager` made it easier for me to extend `deb_pkg_tools.deps` with the changes required to support 'architecture restrictions' (issue #9).
- Add `Build-Depends` to `DEPENDS_LIKE_FIELDS`. I noticed while testing with the example provided in issue #9 that the dependencies in the `Build-Depends` field weren't being parsed. Given that I was working on adding support for parsing of architecture restrictions (as suggested in issue #9) this seemed like a good time to fix this .
- Updated `generate_stdeb_cfg()`.

About backwards compatibility:

I'm bumping the major version number because 754debc0b61 removed the `deb_pkg_tools.compat.total_ordering` and `deb_pkg_tools.utils.OrderedObject` classes and internal methods like `_key()` so strictly speaking this breaks backwards compatibility, however both of these classes were part of miscellaneous scaffolding used by deb-pkg-tools but not an intentional part of the documented API, so I don't expect this to be particularly relevant to most (if not all) users of deb-pkg-tools.

3.1.21 Release 3.1 (2017-01-27)

- Merged pull request #8: Add support for `*.udeb` micro packages.
- Updated test suite after merging #8.
- Suggest memcached in `stdeb.cfg`.
- Added `readme` target to `Makefile`.

3.1.22 Release 3.0 (2016-11-25)

This release was a huge refactoring to enable concurrent related package collection. In the process I switched from SQLite to the Linux file system (augmented by memcached) because SQLite completely collapsed under concurrent write activity (it would crap out consistently beyond a certain number of concurrent readers and writers).

Detailed changes:

- Refactored makefile, setup script, Travis CI configuration, etc.
- Bug fix: Don't unnecessarily garbage collect cache.
- Experimented with increased concurrency using SQLite Write-Ahead Log (WAL).
- Remove redundant :py: prefixes from RST references
- Fix broken RST references logged by `sphinx-build -n`.
- Moved `deb_pkg_tools.utils.compact()` to `humanfriendly.text.compact()`.
- Fixed a broken pretty printer test.
- Implement and enforce PEP-8 and PEP-257 compliance
- Switch from SQLite to filesystem for package cache (to improve concurrency between readers and writers). The WAL did not improve things as much as I would have hoped...
- Document and optimize filesystem based package metadata cache
- Add some concurrency to `deb-pkg-tools --collect` (when more than one archive is given, the collection of related archives is performed concurrently for each archive given).
- Re-implement garbage collection for filesystem based cache.
- Improvements to interactive package collection:
  - Don't use multiprocessing when a single archive is given because it's kind of silly to fork subprocesses for no purpose at all.
  - Restored the functionality of the optional 'cache' argument because the new in memory / memcached / filesystem based cache is so simple it can be passed to multiprocessing workers.
- Enable manual garbage collection (`deb-pkg-tools --garbage-collect`).
- Updated usage in readme.
- Improvements to interactive package collection:
  - A single spinner is rendered during concurrent collection (instead of multiple overlapping spinners that may not be synchronized).
  - The order of the `--collect` and `--yes` options no longer matters.
  - When the interactive spinner is drawn it will always be cleared, even if the operator presses Control-C (previously it was possible for the text cursor to remain hidden after `deb-pkg-tools --collect` was interrupted by Control-C).
- Include command line interface in documentation.

3.1.23 Release 2.0 (2016-11-18)

Stop using the system wide temporary directory in order to enable concurrent builds.

3.1.24 Release 1.37 (2016-11-17)

Significant changes:

- Prefer hard linking over copying of package archives from one directory to another.

- Change Unicode output handling in command line interface. This revisits the 'hack' that I implemented in bc9b52419ea because I noticed today (after integrating `humanfriendly.prompts.prompt_for_confirmation()`) that the wrapping of `sys.stdout` disables libreadline support in interactive prompts (`input()` and `raw_input()`) which means readline hints are printed to stdout instead of being interpreted by libreadline, making interactive prompts rather hard to read :-s.

Miscellaneous changes:

- Test Python 3.5 on Travis CI.
- Don't test tags on Travis CI.
- Use `pip` instead of `python setup.py install` on Travis CI.
- Uncovered and fixed a Python 3 incompatibility in the test suite.

3.1.25 Release 1.36 (2016-05-04)

Make it possible to integrate with GPG agent (`$GPG_AGENT_INFO`).

3.1.26 Release 1.35 (2015-09-24)

Include `Breaks` in control fields parsed like `Depends`.

3.1.27 Release 1.34.1 (2015-09-07)

Bug fix: Invalidate old package metadata caches (from before version 1.31.1).

Should have realized this much sooner of course but I didn't, for which my apologies if this bit anyone like it bit me . I wasted two hours trying to find out why something that was logically impossible (judging by the code base) was happening anyway. Cached data in the old format!

3.1.28 Release 1.34 (2015-07-16)

Automatically embed usage in readme (easier to keep up to date).

3.1.29 Release 1.33 (2015-07-16)

Added `deb_pkg_tools.control.create_control_file()` function.

3.1.30 Release 1.32.2 (2015-05-01)

Bug fixes for related package archive collection.

3.1.31 Release 1.32.1 (2015-05-01)

- Include `Pre-Depends` in control fields parsed like `Depends:`.
- Updated doctest examples with regards to changes in bebe413dcc5.
- Improved documentation of `parse_filename()`.

3.1.32 Release 1.32 (2015-04-23)

Improve implementation and documentation of `collect_related_packages()`.

The result of the old implementation was dependent on the order of entries returned from `os.listdir()` which can differ from system to system (say my laptop vervsus Travis CI) and so caused inconsistently failing builds.

3.1.33 Release 1.31 (2015-04-11)

- Extracted installed version discovery to re-usable function.
- `dpkg-scanpackages` isn't used anymore, remove irrelevant references.

3.1.34 Release 1.30 (2015-03-18)

Added `deb_pkg_tools.utils.find_debian_architecture()` function.

This function is currently not used by deb-pkg-tools itself but several of my projects that build on top of deb-pkg-tools need this functionality and all of them eventually got their own implementation. I've now decided to implement this once, properly, so that all projects can use the same tested and properly documented implementation (as simple as it may be).

3.1.35 Release 1.29.4 (2015-02-26)

Adapted pull request #5 to restore Python 3 compatibility.

3.1.36 Release 1.29.3 (2014-12-16)

Changed SQLite row factory to "restore" Python 3.4.2 compatibility.

The last Travis CI builds that ran on Python 3.4.1 worked fine and no changes were made in deb-pkg-tools since then so this is clearly caused by a change in Python's standard library. This is an ugly workaround but it's the most elegant way I could find to "restore" compatibility.

3.1.37 Release 1.29.2 (2014-12-16)

Bug fix: Don't normalize `Depends:` lines.

Apparently `dpkg-scanpackages` and compatible re-implementations like the one in deb-pkg-tools should not normalize `Depends:` fields because apt can get confused by this. Somehow it uses either a literal comparison of the metadata or a comparison of the hash of the metadata to check if an updated package is available (I tried to find this in the apt sources but failed to do so due to my limited experience with C++). So when the `Depends:` line in the `Packages.gz` file differs from the `Depends:` line in the binary control file inside a `*.deb` apt will continuously re-download and install the same binary package...

3.1.38 Release 1.29.1 (2014-11-15)

Moved `coerce_boolean()` to humanfriendly package.

3.1.39 Release 1.29 (2014-10-19)

Merged pull request #4: Added `$DPT_ALLOW_FAKEROOT_OR_SUDO` and `$DPT_CHOWN_FILES` environment variables to make `sudo` optional.

3.1.40 Release 1.28 (2014-09-17)

Change location of package cache when `os.getuid() == 0`.

3.1.41 Release 1.27.3 (2014-08-31)

Sanitize permissions of `DEBIAN/{pre,post}{inst,rm}` and `etc/sudoers.d/*`.

3.1.42 Release 1.27.2 (2014-08-31)

Improve Python 2.x/3.x compatibility (return lists explicitly).

3.1.43 Release 1.27.1 (2014-08-31)

- Bug fix for SQLite cache string encoding/decoding on Python 3.x.
- Bug fix for check_package() on Python 3.x.
- Bug fix for obscure Python 3.x issue (caused by mutating a list while iterating it).
- Make collect_related_packages() a bit faster (actually quite a lot when `dpkg --compare-versions` is being used).
- Make `deb_pkg_tools.control.*` less verbose.

3.1.44 Release 1.27 (2014-08-31)

- Added command line interface for static checks (with improved test coverage).
- Made `collect_related_packages()` a bit faster.
- "Refine" entry collection strategy for Travis CI.

3.1.45 Release 1.26.4 (2014-08-30)

Restore Python 3.x compatibility (failing build).

3.1.46 Release 1.26.3 (2014-08-30)

Still not enough entropy on Travis CI, let's see if we can work around that...

I tried to fix this using `rng-tools` in 3c372c3097f but that didn't work out due to the way OpenVZ works. This commit introduces a more general approach that will hopefully work on OpenVZ and other virtualized environments, we'll see...

3.1.47 Release 1.26.2 (2014-08-30)

- Restore Python 3 compatibility.
- Improve test coverage.
- Try to work around lack of entropy on Travis CI.

3.1.48 Release 1.26 (2014-08-30)

Add static analysis to detect version conflicts.

3.1.49 Release 1.25 (2014-08-30)

Make `collect_related_packages()` 5x faster:

- Use high performance decorator to memoize overrides of `Relationship.matches()`.
- Exclude conflicting packages from all further processing as soon as they are found.
- Moved the dpkg comparison cache around.
- Removed `Version.__hash__()`.

3.1.50 Release 1.24.1 (2014-08-26)

Bug fix for unused parameter in 442d67cf4dd.

3.1.51 Release 1.24 (2014-08-26)

Normalize setgid bits (because `dpkg-deb` doesn't like them).

3.1.52 Release 1.23.4 (2014-08-04)

Merged pull request #2: Improve platform compatibility with environment variables.

- Added user-name and user-group overrides (`$DPT_ROOT_USER`, `$DPT_ROOT_GROUP`) for systems that don't have a `root` group or when `root` isn't a desirable consideration when building packages.
- Can now disable hard-links (`$DPT_HARD_LINKS`). The `cp -l` parameter is not supported on Mavericks 10.9.2.
- Replaced `du -sB` with `du -sk` (not supported on Mavericks 10.9.2).
- Can now disable `sudo` (`$DPT_SUDO`) since it's sometimes not desirable and not required just to build the package (for example on MacOS, refer to pull request #2 for an actual use case).

3.1.53 Release 1.23.3 (2014-06-27)

Bug fix for `copy_package_files()`.

3.1.54 Release 1.23.2 (2014-06-25)

Further improvements to `collect_packages()`.

3.1.55 Release 1.23.1 (2014-06-25)

Bug fix: Don't swallow keyboard interrupt in `collect_packages()` wrapper.

3.1.56 Release 1.23 (2014-06-25)

Added `group_by_latest_versions()` function.

3.1.57 Release 1.22.6 (2014-06-22)

Try to fix cache deserialization errors on older platforms (refer to the commit message of 8b04dfcd4d3 for more details about the errors I'm talking about).

3.1.58 Release 1.22.5 (2014-06-22)

Preserving Python 2.x *and* Python 3.x compatibility is hard .

3.1.59 Release 1.22.4 (2014-06-22)

Bug fix: Encode stdout/stderr as UTF-8 when not connected to a terminal.

3.1.60 Release 1.22.3 (2014-06-19)

Bug fix for Python 3 syntax compatibility.

3.1.61 Release 1.22.2 (2014-06-19)

Make the package cache resistant against deserialization errors.

Today I've been hitting zlib decoding errors and I'm 99% sure my disk isn't failing (RAID 1 array). For now I'm inclined not to dive too deep into this, because there's a very simple fix (see first line :-). For future reference, here's the zlib error:

```
File ".../deb_pkg_tools/cache.py", line 299, in control_fields
  return self.cache.decode(self['control_fields'])
File ".../deb_pkg_tools/cache.py", line 249, in decode
  return pickle.loads(zlib.decompress(database_value))

error: Error -5 while decompressing data
```

3.1.62 Release 1.22.1 (2014-06-16)

- Change `clean_package_tree()` to clean up `__pycache__` directories.
- Improved test coverage of `check_duplicate_files()`.

3.1.63 Release 1.22 (2014-06-09)

Proof of concept: duplicate files check (static analysis).

3.1.64 Release 1.21 (2014-06-09)

Implement proper package metadata cache using SQLite 3.x (high performance).

I've been working on CPU and disk I/O intensive package analysis across hundreds of package archives which is very slow even on my MacBook Air with four cores and an SSD. I decided to rip the ad-hoc cache in `scan_packages()` out and refactor it into a more general purpose persistent, multiprocess cache implemented on top of SQLite 3.x.

3.1.65 Release 1.20.11 (2014-06-08)

Improve performance: Cache results of `RelationshipSet.matches()`.

3.1.66 Release 1.20.10 (2014-06-08)

Make `deb_pkg_tools.utils.atomic_lock()` blocking by default.

3.1.67 Release 1.20.9 (2014-06-07)

Make it possible to ask a `RelationshipSet` for all its names.

3.1.68 Release 1.20.8 (2014-06-07)

Bug fix for Python 3.x compatibility.

3.1.69 Release 1.20.7 (2014-06-07)

Sanitize permission bits of root directory when building packages.

3.1.70 Release 1.20.6 (2014-06-07)

Switch to executor 1.3 which supports `execute(command, fakeroot=True)`.

3.1.71 Release 1.20.5 (2014-06-05)

Added `deb_pkg_tools.control.load_control_file()` function.

3.1.72 Release 1.20.4 (2014-06-01)

Minor optimization that seems to make a major difference (without this optimization I would sometimes hit "recursion depth exceeded" errors).

3.1.73 Release 1.20.3 (2014-06-01)

Bug fix for Python 3.x compatibility (missed `compat.basestring` import).

3.1.74 Release 1.20.2 (2014-06-01)

Bug fix for Python 3.x incompatible syntax in newly added code.

3.1.75 Release 1.20.1 (2014-06-01)

Automatically create parent directories in `atomic_lock` class.

3.1.76 Release 1.20 (2014-06-01)

Re-implemented `dpkg-scanpackages -m` in Python to make it really fast.

3.1.77 Release 1.19 (2014-06-01)

Added function `deb_pkg_tools.package.find_package_archives()`.

3.1.78 Release 1.18.5 (2014-05-28)

Bug fix for `find_latest_version()` introduced in commit 5bf01b0 (build failure on Travis CI).

3.1.79 Release 1.18.4 (2014-05-28)

Disable pretty printing of `RelationshipSet` objects by default.

3.1.80 Release 1.18.3 (2014-05-26)

- Fixed sort order of `deb_pkg_tools.package.PackageFile` (changed field order)
- Sanity check given arguments in `deb_pkg_tools.package.find_latest_version()`.
- Documented the exception that can be raised by `deb_pkg_tools.package.parse_filename()`.

3.1.81 Release 1.18.2 (2014-05-26)

Change `deb_pkg_tools.deps.parse_depends()` to accept a list of dependencies.

3.1.82 Release 1.18.1 (2014-05-25)

- Bug fix for last commit (avoid `AttributeError` on `apt_pkg.version_compare`).
- Changed documentation of `deb_pkg_tools.compat` module.
- Made doctest examples Python 3.x compatible (`print()` as function).
- Integrate customized doctest checking in makefile.

3.1.83 Release 1.18 (2014-05-25)

Extract version comparison to separate module (with tests).

I wanted to re-use version sorting in several places so it seemed logical to group the related code together in a new `deb_pkg_tools.version` module. While I was at it I decided to write tests that make sure the results of `compare_versions_with_python_apt()` and `compare_versions_with_dpkg()` are consistent with each other and the expected behavior.

3.1.84 Release 1.17.7 (2014-05-18)

Made `collect_related_packages()` faster (by splitting `inspect_package()`).

3.1.85 Release 1.17.6 (2014-05-18)

Re-implemented `dpkg_compare_versions()` on top of `apt.VersionCompare()`.

3.1.86 Release 1.17.5 (2014-05-18)

Moved Python 2.x / 3.x compatibility functions to a separate module.

3.1.87 Release 1.17.4 (2014-05-18)

- Made pretty print tests compatible with Python 3.x.
- Removed `binutils` and `tar` dependencies (these are no longer needed since the `inspect_package()` function now uses the `dpkg-deb` command).

3.1.88 Release 1.17.3 (2014-05-18)

- Cleanup pretty printer, remove monkey patching hack, add tests.
- Dedent string passed to `deb822_from_string()` (nice to use in tests).

3.1.89 Release 1.17.2 (2014-05-18)

- Bug fix for output of `deb-pkg-tools --inspect ...` (broken in Python 3.x compatibility spree).
- Monkey patch pprint so it knows how to 'pretty print' `RelationshipSet` (very useful to verify docstrings containing doctest blocks).
- Improved test coverage of `deb_pkg_tools.package.PackageFile.__lt__()`.

3.1.90 Release 1.17.1 (2014-05-18)

- Bug fix for `deb_pkg_tools.deps.parse_relationship()`.
- Bug fix for `inspect_package()` (hard links weren't recognized).
- Added `deb_pkg_tools.control.deb822_from_string()` shortcut.
- Various bug fixes for Python 2.6 and 3.x compatibility:

- Bumped `python-debian` requirement to `0.1.21-nmu2` for Python 3.x compatibility
- Changed `logger.warn()` to `logger.warning()` (the former is deprecated).
- Fixed missing `str_compatible` decorator (Python 3.x compatibility).

3.1.91 Release 1.17 (2014-05-18)

Added `collect_related_packages()` function and `deb-pkg-tools --collect` command line interface.

3.1.92 Release 1.16 (2014-05-18)

- Added relationship parsing/evaluation module (`deb_pkg_tools.deps.*`).
- Bug fix for `deb_pkg_tools.generate_stdeb_cfg()`.
- Test suite changes:
  - Skip repository activation in `test_command_line_interface()` when not `root`.
  - Added an improvised slow test marker.

3.1.93 Release 1.15.2 (2014-05-16)

- Added `deb_pkg_tools.package.parse_filename()` function.
- Properly document `deb_pkg_tools.package.ArchiveEntry` named tuple.
- Improved test coverage by testing command line interface.
- Changed virtual environment handling in `Makefile`.

3.1.94 Release 1.15.1 (2014-05-10)

- Bug fix for Python 3 compatibility.
- Moved `deb_pkg_tools.cli.with_repository()` to `deb_pkg_tools.repo.with_repository()`.
- Submit test coverage from travis-ci.org to coveralls.io, add dynamic coverage statistics to `README.rst`.
- Run more tests on travis-ci.org by running test suite as root (this gives the test suite permission to test things like apt-get local repository activation).
- Improved test coverage of `deb_pkg_tools.repository.update_repository()` and `deb_pkg_tools.gpg.GPGKey()`.

3.1.95 Release 1.15 (2014-05-10)

- Merge pull request #1: Python 3 compatibility.
- Document supported Python versions (2.6, 2.7 & 3.4).
- Start using travis-ci.org to avoid dropping Python 3 compatibility in the future.
- Update documented dependencies in `README.rst`.

3.1.96 Release 1.14.7 (2014-05-04)

Refactored `deb_pkg_tools.utils.execute()` into a separate package.

3.1.97 Release 1.14.6 (2014-05-03)

Bug fix for globbing support.

3.1.98 Release 1.14.5 (2014-05-03)

Added support for `deb-pkg-tools --patch=CTRL_FILE --set="Name: Value"`.

3.1.99 Release 1.14.4 (2014-05-03)

Make `update_repository()` as "atomic" as possible.

3.1.100 Release 1.14.3 (2014-05-03)

Support for globbing in configuration file (`repos.ini`).

3.1.101 Release 1.14.2 (2014-04-29)

Bug fix: Typo in readme (found just after publishing of course).

3.1.102 Release 1.14.1 (2014-04-29)

Added support for the system wide configuration file `/etc/deb-pkg-tools/repos.ini`.

3.1.103 Release 1.14 (2014-04-29)

- Make repository generation user configurable (`~/.deb-pkg-tools/repos.ini`).
- Test GPG key generation (awkward but useful, make it opt-in or opt-out?).
- Make Python >= 2.6 dependency explicit in stdeb.cfg (part 2 :-).
- Documentation bug fix: Update usage message and `README.rst`.

3.1.104 Release 1.13.2 (2014-04-28)

Bug fix: Respect the `build_package(copy_files=False)` option.

3.1.105 Release 1.13.1 (2014-04-28)

- Try to detect removal of `*.deb` files in `update_repository()`.
- Bring test coverage back up to >= 90%.

3.1.106 Release 1.13 (2013-11-16)

Make `inspect_package()` report package contents. This was added to make it easier to write automated tests for deb-pkg-tools but may be useful in other circumstances and so became part of the public API .

3.1.107 Release 1.12.1 (2013-11-03)

Make Python >= 2.6 dependency explicit in `stdeb.cfg`.

3.1.108 Release 1.12 (2013-11-03)

Make `copy_package_files()` more generally useful.

3.1.109 Release 1.11 (2013-11-02)

- Improve `deb_pkg_tools.gpg.GPGKey` and related documentation.

3.1.110 Release 1.10.2 (2013-11-02)

Bug fix: Make `update_repository()` always remove old `Release.gpg` files.

3.1.111 Release 1.10.1 (2013-11-02)

Bug fix: Make `update_repository()` fully aware of `apt_supports_trusted_option()`.

3.1.112 Release 1.10 (2013-11-02)

Use the `[trusted=yes]` option in `sources.list` when possible:

With this we no longer need a generated GPG key at all; we just skip all steps that have anything to do with GPG :-). Unfortunately we still need to be backwards compatible so the code to generate and manage GPG keys remains for now...

3.1.113 Release 1.9.9 (2013-10-22)

Remove automatic dependency installation (too much magic, silly idea).

3.1.114 Release 1.9.8 (2013-10-22)

Bug fixes for last commit (sorry about that!).

3.1.115 Release 1.9.7 (2013-10-22)

New `deb-pkg-tools --with-repo=DIR COMMAND...` functionality (only exposed in the command line interface for now).

3.1.116 Release 1.9.6 (2013-10-21)

Workaround for old and buggy versions of GnuPG .

3.1.117 Release 1.9.5 (2013-10-20)

Bug fix for `update_repository()`.

3.1.118 Release 1.9.4 (2013-10-20)

Change `update_repository()` to only rebuild repositories when contents have changed.

3.1.119 Release 1.9.3 (2013-10-20)

Make `update_conffiles()` work properly in Python < 2.7.

3.1.120 Release 1.9.2 (2013-10-20)

Enable overriding of GPG key used by the `deb_pkg_tools.repo.*` functions.

3.1.121 Release 1.9.1 (2013-10-20)

Made it possible not to copy the files in the build directory (`build_package()`).

3.1.122 Release 1.9 (2013-10-20)

Extracted GPG key generation into standalone function.

3.1.123 Release 1.8 (2013-10-20)

Automatic installation of required system packages.

3.1.124 Release 1.7.2 (2013-10-19)

Make `copy_package_files()` compatible with `schroot` environments.

3.1.125 Release 1.7.1 (2013-10-18)

Enable callers of `update_repository()` to set fields of `Release` files.

3.1.126 Release 1.7 (2013-10-16)

Change `build_package()` to automatically update `DEBIAN/conffiles`.

3.1.127 Release 1.6.2 (2013-10-13)

Bug fix: Make `deb-pkg-tools -u` and `deb-pkg-tools -a` compatible with `schroot` environments.

3.1.128 Release 1.6.1 (2013-10-12)

Added `stdeb.cfg` to `MANIFEST.in`.

3.1.129 Release 1.6 (2013-10-12)

- Improved documentation of `deb_pkg_tools.utils.execute()`.
- Improved `deb_pkg_tools.utils.execute()`, implemented optional `sudo` support.

3.1.130 Release 1.5 (2013-10-12)

Automatically generate a GPG automatic signing key the first time it's needed.

3.1.131 Release 1.4.3 (2013-10-12)

- Made log messages more user friendly.
- Made Debian package dependencies available from Python.

3.1.132 Release 1.4.2 (2013-10-12)

Make it possible to delete fields using `patch_control_file()`.

3.1.133 Release 1.4.1 (2013-08-13)

Improved `update_installed_size()` (by using `patch_control_file()`).

3.1.134 Release 1.4 (2013-08-13)

Normalize field names in control files (makes merging easier).

3.1.135 Release 1.3.2 (2013-08-13)

Make `build_package()` sanitize file modes:

I was debating with myself for quite a while how far to go in these kinds of "sensible defaults"; there will always be someone who doesn't want the behavior. I decided that those people shouldn't be using deb-pkg-tools then :-) (I wonder how long it takes though, before I find myself in that group of people ;-).

3.1.136 Release 1.3.1 (2013-08-11)

Improved `clean_package_tree()` (better documentation, more files to ignore).

3.1.137 Release 1.3 (2013-08-11)

Added `clean_package_tree()` function.

3.1.138 Release 1.2 (2013-08-10)

Added `patch_control_file()` function.

3.1.139 Release 1.1.4 (2013-08-10)

Removed as much manual shell quoting as possible.

3.1.140 Release 1.1.3 (2013-08-10)

- Silenced `deb_pkg_tools.utils.execute()`
- Simplified `deb_pkg_tools.package.inspect_package()`.

3.1.141 Release 1.1.2 (2013-08-07)

Started using `coloredlogs.increase_verbosity()`.

3.1.142 Release 1.1.1 (2013-08-07)

Loosen up the requirements (stop using absolute version pinning).

3.1.143 Release 1.1 (2013-08-05)

Automatically run Lintian after building packages.

3.1.144 Release 1.0.3 (2013-08-04)

Improved wording of readme, fixed typo in docs.

3.1.145 Release 1.0.2 (2013-08-04)

Got rid of the use of shell pipes in order to detect "command not found" errors.

3.1.146 Release 1.0.1 (2013-08-04)

Brought test suite coverage up to 96% .

3.1.147 Release 1.0 (2013-07-26)

Initial commit with a focus on:

- Building of Debian binary packages.
- Inspecting the metadata of Debian binary packages.
- Creation of trivial repositories based on collected package metadata.

Python Module Index

d

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W