Version Tools Documentation

Release 1.9.1

Zygmunt Krynicki

January 03, 2014

Contents

See Also:

To get started quickly see *Usage instructions*

See Also:

See what's new in *Version 1.9.1*

Note: This document may be out of date, the bleeding edge version is always available at http://versiontools.rtfd.org/

CHAPTER 1

Installation

This package is being actively maintained and published in the Python Package Index. You can install it if you have pip tool using just one line:

```
$ pip install versiontools
```

CHAPTER 2

Features

- A piece of code that allows you to keep a **single version definition** inside your package or module. No more hacks in setup.py, no more duplicates in setup.py and somewhere else. Just **one** version per package.
- `Version` objects can produce nice version strings for released files that are compliant with **PEP 386**. Releases, alphas, betas, development snaphots. All of those get good version strings out of the box.
- Version objects understand the VCS (Version Control System) used by your project. Git, Mercurial and Bazaar are supported out of the box. Custom systems can be added by 3rd party plugins.
- Version object that compares as a tuple of values and sorts properly.
- Zero-dependency install! If all you care about is handling setup() to get nice tarball names then you don't need to depend on versiontools (no setup_requires, no install_requires!). You will need to bundle a small support module though.

CHAPTER 3

__version__ to string conversion

This is pulled from the documentation of the string method on the `Version` class. In general you don't need to explicitly use that class to benefit from this system. To learn more check the *Usage instructions* section.

`Version.__str__()`

Return a string representation of the version tuple.

The string is not a direct concatenation of all version components. Instead it's a more natural 'human friendly' version where components with certain values are left out.

The following table shows how a version tuple gets converted to a version string.

__version__	Formatter version
`(1, 2, 0, "final", 0)`	`"1.2"`
`(1, 2, 3, "final", 0)`	`"1.2.3"`
`(1, 3, 0, "alpha", 1)`	`"1.3a1"`
`(1, 3, 0, "beta", 1)`	`"1.3b1"`
`(1, 3, 0, "candidate", 1)`	`"1.3c1"`
`(1, 3, 0, "dev", 0)`	`"1.3.dev"`

Now when release level is set to `"dev"` then interesting things start to happen. When possible, version control system is queried for revision or changeset identifier. This information gets used to create a more useful version string. The suffix gets appended to the base version string. So for example a full version string, when using Bazaar might look like this: `"1.3.dev54"` which indicates that the tree was at revision 54 at that time.

The following table describes what gets appended by each version control system.

VCS	Formatted version suffix
Bazaar	Revision number (revno), e.g. `54`
Git	Short commit ID of the current branch e.g. `"763fbe3"`
Mercurial	Tip revision number, e.g. `54`

Note: This logic is implemented in `versiontools.Version.__str__()` and can be overridden by sub-classes. You can use project-specific logic if required. To do that replace __version__ with an instance of your sub-class.

CHAPTER 4

Indices and tables

4.1 Usage instructions

Using versiontools is very easy. Just follow those steps to do it.

4.1.1 Declare package version

Put this code your package's `__init__.py` or in your main module:

```
__version__ = (1, 2, 3, 'final', 0)  # replace with your project version
```

4.1.2 Copy versiontools_support.py

You will need to keep a copy of `versiontools_support.py` file in your source tree. This file will be needed by your users that don't have versiontools to still be able to run setup.py to install your package.

4.1.3 Update MANIFEST.in

You will need to update (or create) `MANIFEST.in` to ensure that each source distribution you make with `setup.py sdist` will ship a copy of the support file. All that you have to do is to append this line to your `MANIFEST.in`:

```
include versiontools_support.py
```

4.1.4 Patch setup.py

Edit your `setup.py` to have code that looks like this:

```
import versiontools_support

setup(
    version = ":versiontools:your_package",
)
```

The trick here is to use a magic value for version keyword. The format of that magic value is:

```
":versiontools:" - a magic string that versiontools matches
your_package     - name of your package or module to import
":"              - colon separating package from identifier (optional)
identifier       - Object to import from your_package. (optional)
                   Can be omitted if equal to __version__.
```

This will make versiontools use `versiontools.format_version()` on whatever *your_package.__version__* contains. Since your *__version__* is a tuple as described above you'll get a correctly formatted version identifier.

This code will ensure that:

1. Your package will use version tools
2. Your package will correctly install via pip

Developing with versiontools

While you are working on the next version of your project you should make sure that `releaselevel` is set to `"dev"`. This will (if you have proper vcs integration in place) allow you to get the most benefit.

Releases

Each time you make a release (with setup.py sdist or any bdist commands) make sure to change the `releaselevel` to something other than `"dev"`. You can use the following strings:

- `"alpha"`
- `"beta"`
- `"candidate"`
- `"final"`

4.2 Integration with version control

versiontools supports a form of version control system integration. This code is *only* triggered for development versions of your project (indicated by setting `releaselevel` to `dev"`)

In development mode the generated version string will include the revision number or the abbreviated hash of the current commit. This makes it nice for ongoing releases on pypi as after each commit your source tarballs will be different.

4.2.1 Batteries included

The following version control systems are supported out of the box. To use them you need to have the corresponding libraries installed. Check the links below for details.

Bazaar

Using bazaar appends the branch revision to the version string. See *versiontools.bzr_support*

Git

Using git appends the short commit id of the active branch. See *versiontools.git_support*

Mercurial

Mercurial plug-in appends the branch revision to the development version. See *versiontools.hg_support*

4.2.2 Custom version control systems

VCS integration is not hard-coded into versiontools. Instead any package that uses setuptools and provides an entry point `versiontools.vcs_integration` can add support for integration with additional version control systems.

To see how to implement this simple API refer to the bundled plug-in for Bazaar `BzrIntegration`, Git `GitIntegration` or Mercurial `HgIntegration`.

To make versiontools aware of additional plug-ins they need to be registered in the entry points database. To do that make sure your package uses setuptools and put the following snippet into your `setup.py`:

```
setup(
    name="foo",
    description="The imaginary foo version control system",
    entry_points="""
    [versiontools.vcs_integration]
    foo=foo.versiontools_plugin:FooIntegration
    """
    )
```

This will make versiontools look for the `foo` system by importing `foo.versiontools_plugin` and extracting the `FooIntegration` class. Remember that your `foo` package needs to be installed for this system to work.

4.3 Code reference

4.3.1 versiontools

Define *single* and *useful* `__version__` of a project.

class `versiontools.`**`Version`**
Smart version class.

Version class is a tuple of five elements and has the same logical components as `sys.version_info`.

In addition to the tuple elements there is a special `vcs` attribute that has all of the data exported by the version control system.

static `__new__`(*major*, *minor*, *micro=0*, *releaselevel='final'*, *serial=0*)
Construct a new version tuple.

There is some extra logic when initializing tuple elements. All variables except for releaselevel are silently converted to integers That is:

```
>>> Version("1.2.3.dev".split("."))
(1, 2, 3, "dev", 0)
```

Parameters

- **major** (`int` or `str`) – Major version number
- **minor** (`int` or `str`) – Minor version number
- **micro** (`int` or `str`) – Micro version number, defaults to `0`.
- **releaselevel** (`str`) – Release level name.

  There is a constraint on allowed values of releaselevel. Only the following values are permitted:

  – 'dev'

  – 'alpha'

  – 'beta'

  – 'candidate'

  – 'final'
- **serial** – Serial number, usually zero, only used for alpha, beta and candidate versions where it must be greater than zero.

Raises ValueError If releaselevel is incorrect, a version component is negative or serial is 0 and releaselevel is alpha, beta or candidate.

major
Major version number

minor
Minor version number

micro
Micro version number

releaselevel
Release level string

serial
Serial number

classmethod from_tuple(*version_tuple*)
Create version from 5-element tuple

Note: This method is identical to the constructor, just spelled in a way that is more obvious to use.

New in version 1.1.

classmethod from_tuple_and_hint(*version_tuple*, *hint*)
Create version from a 5-element tuple and VCS location hint.

Similar to `from_tuple()` but uses the hint object to locate the source tree if needed. A good candidate for hint object is the module that contains the version_tuple. In general anything that works with `inspect.getsourcefile()` is good. New in version 1.4.

classmethod from_expression(*pkg_expression*)
Create a version from a python module name.

The argument must describe a module to import. The module must declare a variable that holds the actual version. The version cannot be a plain string and instead must be a tuple of five elements as described by the `Version` class.

The variable that holds the version should be called `__version__`. If it is called something else the actual name has to be specified explicitly in `pkg_expression` by appending a colon (`:`) and the name of the variable (for example `package:version`). New in version 1.9.

vcs

Return VCS integration object, if any.

Accessing this attribute for the first time will query VCS lookup (may be slower, will trigger imports of various VCS plugins).

The returned object, if not None, should have at least the *revno* property. For details see your particular version control integration plugin.

Note: This attribute is **not** an element of the version tuple and thus does not break sorting.

New in version 1.0.4.

__str__()

Return a string representation of the version tuple.

The string is not a direct concatenation of all version components. Instead it's a more natural 'human friendly' version where components with certain values are left out.

The following table shows how a version tuple gets converted to a version string.

__version__	Formatter version
`(1, 2, 0, "final", 0)`	`"1.2"`
`(1, 2, 3, "final", 0)`	`"1.2.3"`
`(1, 3, 0, "alpha", 1)`	`"1.3a1"`
`(1, 3, 0, "beta", 1)`	`"1.3b1"`
`(1, 3, 0, "candidate", 1)`	`"1.3c1"`
`(1, 3, 0, "dev", 0)`	`"1.3.dev"`

Now when release level is set to `"dev"` then interesting things start to happen. When possible, version control system is queried for revision or changeset identifier. This information gets used to create a more useful version string. The suffix gets appended to the base version string. So for example a full version string, when using Bazaar might look like this: `"1.3.dev54"` which indicates that the tree was at revision 54 at that time.

The following table describes what gets appended by each version control system.

VCS	Formatted version suffix
Bazaar	Revision number (revno), e.g. `54`
Git	Short commit ID of the current branch e.g. `"763fbe3"`
Mercurial	Tip revision number, e.g. `54`

versiontools.**format_version**(*version*, *hint=None*)

Pretty formatting for 5-element version tuple.

Instead of using `Version` class directly you may want to use this simplified interface where you simply interpret an arbitrary five-element version tuple as a version to get the pretty and **PEP 386**-compliant version string.

Parameters

- **version** (A `tuple` with five elements, as the one provided to `versiontools.Version.from_tuple()`, or an existing instance of `versiontools.Version`.) – The version to format

- **hint** (either `None`, or a module.) – The hint object, if provided, helps versiontools to locate the directory which might host the project's source code. The idea is to pass *module.__version__* as the first argument and *module* as the hint. This way we can lookup where module came from, and look for version control system data in that directory. Technically passing hint will make us call `from_tuple_and_hint()` instead of `from_tuple()`.

New in version 1.1.

4.3.2 versiontools.setuptools_hooks

Plugins for setuptools that add versintools features.

Setuptools has a framework where external packages, such as versiontools, can hook into setup.py metadata and commands. We use this feature to intercept special values of the `version` keyword argument to `setup()`. This argument handled by the following method:

`versiontools.setuptools_hooks.`**`version`**(*dist*, *attr*, *value*)
Handle the `version` keyword to setuptools.setup()

Note: This function is normally called by setuptools, it is advertised in the entry points of versiontools as setuptools extension. There is no need to call in manually.

New in version 1.3.

4.3.3 versiontools.versiontools_support

A small standalone module that allows any package to use versiontools.

Typically you should copy this file verbatim into your source distribution.

Historically versiontools was depending on a exotic feature of setuptools to work. Setuptools has so-called setup-time dependencies, that is modules that need to be downloaded and imported/interrogated for setup.py to run successfully. Versiontools supports this by installing a handler for the 'version' keyword of the setup() function.

This approach was always a little annoying as this setuptools feature is rather odd and very few other packages made any use of it. In the future the standard tools for python packaging (especially in python3 world) this feature may be removed or have equivalent thus rendering versiontools completely broken.

Currently the biggest practical issue is the apparent inability to prevent setuptools from downloading packages designated as setup_requires. This is discussed in this pip issue: https://github.com/pypa/pip/issues/410

To counter this issue I've redesigned versiontools to be a little smarter. The old mode stays as-is for compatibility. The new mode works differently, without the need for using setup_requires in your setup() call. Instead it requires each package that uses versiontools to ship a verbatim copy of this module and to import it in their setup.py script. This module helps setuptools find package version in the standard PKG-INFO file that is created for all source distributions. Remember that you only need this mode when you don't want to add a dependency on versiontools. This will still allow you to use versiontools (in a limited way) in your setup.py file.

Technically this module defines an improved version of one of distutils.dist.DistributionMetadata class and monkey-patches distutils to use it. To retain backward compatibility the new feature is only active when a special version string is passed to the setup() call.

class `versiontools.versiontools_support.`**`VersiontoolsEnchancedDistributionMetadata`**(*path=None*)
A subclass of distutils.dist.DistributionMetadata that uses versiontools

Typically you would not instantiate this class directly. It is constructed by distutils.dist.Distribution.__init__() method. Since there is no other way to do it, this module monkey-patches distutils to override the original version of DistributionMetadata

get_version()
Get distribution version.

This method is enhanced compared to original distutils implementation. If the version string is set to a special value then instead of using the actual value the real version is obtained by querying versiontools.

If versiontools package is not installed then the version is obtained from the standard section of the `PKG-INFO` file. This file is automatically created by any source distribution. This method is less useful as it cannot take advantage of version control information that is automatically loaded by versiontools. It has the advantage of not requiring versiontools installation and that it does not depend on `setup_requires` feature of `setuptools`.

4.3.4 versiontools.bzr_support

Bazaar support for versiontools

Note: To work with Bazaar repositories you will need bzrlib. You can install it with pip or from the `bzr` package on Ubuntu.

Warning: On Windows the typical Bazaar installation bundles both the python interpreter and a host of libraries and those libraries are not accessible by the typically-installed python interpreter. If you wish to use Bazaar on windows we would recommend to install Bazaar directly from pypi.

class versiontools.bzr_support.**BzrIntegration**(*branch*)
Bazaar integration for versiontools

branch_nick
Nickname of the branch New in version 1.0.4.

classmethod from_source_tree(*source_tree*)
Initialize `BzrIntegration` by pointing at the source tree. Any file or directory inside the source tree may be used.

revno
Revision number of the branch

4.3.5 versiontools.git_support

Git support for versiontools

Note: To work with Git repositories you will need GitPython. Version 0.1.6 is sufficient to run the code. You can install it with pip.

---

class versiontools.git_support.**GitIntegration**(*repo*)
Git integration for versiontools

branch_nick
Nickname of the branch New in version 1.0.4.

commit_id
The full commit id

commit_id_abbrev
The abbreviated, 7 character commit id

classmethod from_source_tree(*source_tree*)
Initialize `GitIntegration` by pointing at the source tree. Any file or directory inside the source tree may be used.

revno
Same as `commit_id_abbrev`

4.3.6 versiontools.hg_support

Mercurial (Hg) support for versiontools.

Note: To work with Mercurial repositories you will need Mercurial. You can install it with pip or from the *mercurial* package on Ubuntu.

---

class `versiontools.hg_support.`**HgIntegration**(*repo*)
Hg integration for versiontools

branch_nick
Nickname of the branch New in version 1.0.4.

classmethod from_source_tree(*source_tree*)
Initialize `HgIntegration` by pointing at the source tree. Any file or directory inside the source tree may be used.

revno
Revision number of the branch

4.4 Release history

4.4.1 Version 1.9.1

- Just bump version to final, sorry

4.4.2 Version 1.9

- Reorganize and update documentation.
- Add a new way of using versiontools that does not require using `setup_requires`. This way is based on bundling a small helper module to help you bootstrap your project when installed from source.
- Add `versiontools.Version.from_expression()` that creates a Version object from a python import expression (and an optional variable identifer)
- Move and rename private function `versiontools.handle_version` to `versiontools.setuptools_hooks:version()`.
- Move and rename private function `versiontools.get_exception_message` to `versiontools._get_exception_message()`.

- Remove private function `versiontools.isstring`.

4.4.3 Version 1.8.3

- Fix incorrectly specified line in git support. Previosuly a KeyError may bleed to the outside calling code, depending on python version.
- Fix incorrectly specified line in bzr support. Previously a non-bzr directory could be associated as a malformed bzr branch.

4.4.4 Version 1.8.2

- Improve git support by adding code paths compatible with python-git 0.1.6 (which is easier to get on Debian)
- Change git support to default to short commit id. If you want to access the long commit id you need to access it directly as GitIntegration.commit_id

4.4.5 Version 1.8.1

- Improve performance when working with checkouts. The use of branch.nick has been replaced with branch._get_nick(local=True). This avoids network operations and is much more responsive.

4.4.6 Version 1.8

- Fixed all pep8 issues (prettier syntax)
- Fixed an issue with using __import__ on Python 2.4.
- Fixed an issue with using :versiontools:path.to.symbol with nested modules
- Fixed an issue with using exceptions on Python 3.x
- Added test that demostrated that exception handling works on all Python versions.
- Unified error handling across version control plugins.
- General documentation improvements, installation, usage, code refrerence, and writing additional plug-ins.
- Added a *backwards incompatible* constraint on serial to be greater than zero on alpha, beta and release candidates as required by **PEP 386**.

4.4.7 Version 1.7

- Add support for Mercurial repositories
- Fix a bug in exception handling that affected 1.6

4.4.8 Version 1.6

- Add support for python2.4 and python2.5 thanks to Jannis Leidel (thanks!).
- Add tox (http://codespeak.net/~hpk/tox/) configuration file for easier testing.

4.4.9 Version 1.5

- Added Git support, contributed by Jannis Leidel (thanks!). To use it you need GitPython >= 0.3.2.RC1. It does not currently work on GitPython packaged in Ubuntu Natty (0.1.6).

4.4.10 Version 1.4

- Work harder to figure out the source tree a __version__ object comes from. This is possible with a new function `versiontools.Version.from_tuple_and_hint()`.
- Allow people to omit the version identifier in setup.py (defaulting to `__version__`)

4.4.11 Version 1.3.2

- Change version string produced by `versiontools.Version.__str__()` and *versiontools.format_version* to be more useful when vcs integration is not available and the release is not final. Consult the table below for details.

Prior to 1.3.2	1.3.2	Comment
`1.2.3` When VCS integration is not available	`1.2.3.dev`	When releaselevel==dev but VCS integration is not available we now add a .dev suffix to differentiate from released versions
`1.2.3a5` or `1.2.3a5.devREVNO` `1.2.3b5` or `1.2.3b5.devREVNO`	`1.2.3a5` ‘ `1.2.3b5`	It will never appear on alphas , betas or release candidates.
`1.2.3c5` or `1.2.3c5.devREVNO`	`1.2.3c5`	

4.4.12 Version 1.3.1

- To make *setup.py test* work in third party components we cannot use versiontools in our own setup.

4.4.13 Version 1.3

- Add integration with setuptools (or more accurately, distribute) so that you no longer have to try-import versiontools. This means that you may finally install your packages with pip and everything will work correctly.
- Prevent an unexplained crash when following the backtrace in `Version._find_source_tree()`.

See Also:

To get started quickly see *Usage instructions*

4.4.14 Version 1.2

- Change how vcs objects are constructed. With this change they are only constructed lazily when needed. This speeds up common operations, delays the time any additional modules are imported (if any) and retains backwards comp ability.
- Updated documentation on installation instructions to point to the new PPA

- Updated recommended usage guide so that installed programs do not depend on versiontools. This allows you to use versiontools in setup.py and still benefit from the smart version formatting and keep your deployment lightweight.
- Added basic unit tests
- Fixed most PEP8 issues

4.4.15 Version 1.1

- Change version string produced by `versiontools.Version.__str__()` to be compatible with **PEP 386** The following table shows how old versions map to new versions:

Old Version	New Version	Comment
`1.2`	`1.2`	
`1.2.3`	`1.2.3`	
`1.2.3.dev.5`	`1.2.3.devREVNO`	VCS revision and serial are two distinct fields. Serial is no longer displayed for development releases.
`1.2.3.alpha.5`	`1.2.3a5` or `1.2.3a5.devREVNO`	`.devREVNO` is only added when VCS integration is available.
`1.2.3.beta.5`	`1.2.3b5` or `1.2.3b5.devREVNO`	
`1.2.3.candidate.5`	`1.2.3c5` or `1.2.3c5.devREVNO`	

- Add `versiontools.format_version()` that converts a 5-element tuple to a proper version string and is more obvious in intent.
- Change default of Version releaselevel to “final”
- Change default of Version serial to 0
- Serial field is no longer initialized with revision number from vcs, instead it is used to count alphas, betas and release candidates.
- All version components except for releaselevel must be non-negative integers or strings that can be converted to such integers
- Do not warn about “directory foo is not a bzr branch”. This message was changed to debug as it is now legitimate for released code not to have bzr version control files.

4.4.16 Version 1.0.4

- Add support to obtain VCS integration object via `vcs` attribute
- Add support to obtain branch nickname from `BzrIntegration` (via `branch_nick` property)
- Add *Code reference*.

4.4.17 Version 1.0.3

- Don’t crash when `ImportError` occurs during VCS integration initialization

4.4.18 Version 1.0.2

- Add documentation
- Fix chicken-and-egg problem so that packages can now depend on versiontools and still be installed correctly with pip

4.4.19 Version 1.0.1

- Make VCS integration more robust in the way it locates source tree

4.4.20 Version 1.0

- Initial release
- *genindex*
- *modindex*
- *search*

Python Module Index

V