versionah

Release 0.16.0

Nov 07, 2019

Contents

versionah is a simple tool to help you — or more specifically *me* — easily maintain version information for a project. Its entire aim is to make the act of displaying or bumping a project's version number a thoughtless task.

All repetitive tasks should be easily automated, and **versionah** is just a little step in the right direction!

It is written in Python, and requires v3.5 or later. **versionah** is released under the GPL v3

Git repository https://github.com/JNRowe/versionah/

Issue tracker https://github.com/JNRowe/versionah/issues/

Contributors https://github.com/JNRowe/versionah/contributors/

CHAPTER 1

Contents

1.1 Background

I maintain many projects, both my toy stuff on GitHub and more serious things at the office. While skipping around in my editor to increase a version number I came to the blindingly obvious realisation that I shouldn't be doing this manually.

Bumping or querying version numbers should be a zero thought process. I shouldn't need to remember the REGEX (Regular Expression) needed to make my editor jump to the version identifier in any given file. I shouldn't need to resort to various `C-a` and `C-x` contortions in vim or formulating complicated lisp functions with `number-to-string` and `string-to-number` in emacs.

Now **`versionah`** is born, and I should be able to realise those dreams!

1.2 Version numbers

This — for some — is a very complicated topic, but not for me. Version numbers are made of three components; major, minor and micro. All three components are natural numbers; no exceptions.

If you find version numbers like 0.6c11 acceptable then **`versionah`** is not for you.

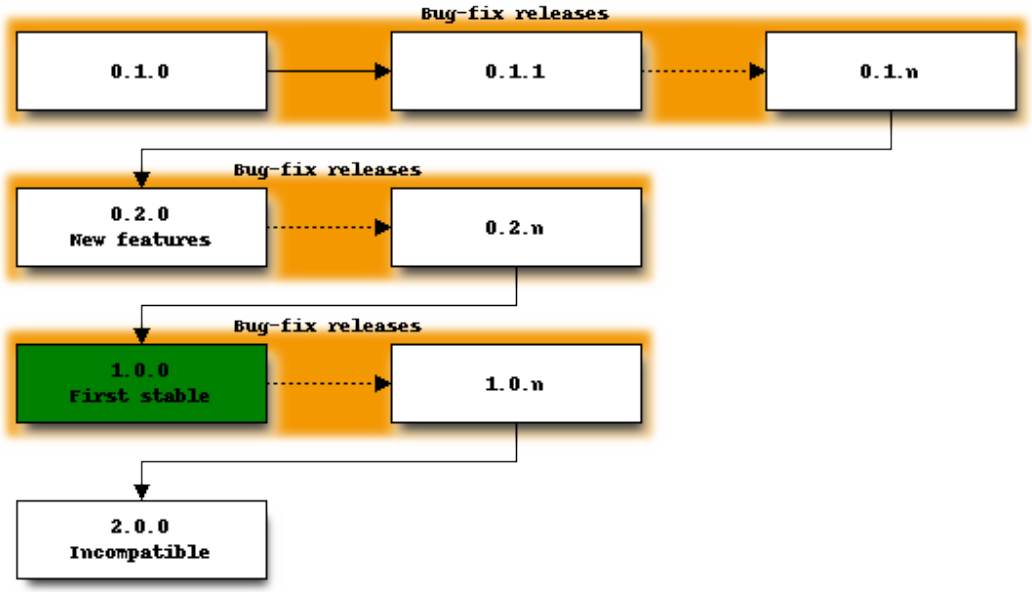
Note: If you like version numbers with two or four integer components then **`versionah`** can be for you too. Support was added in 0.6.0, but that doesn't mean you have to use it!

1.2.1 PEP 386

The version numbering scheme supported by **`versionah`** is a very small subset of `LooseVersion` defined in **PEP 386**. It isn't compliant with `StrictVersion` because of the 4 component support, but support for packages in the wild is much more important to me.

Versioning policy

Beyond the simple rule above you're free to do as you wish, but consider this a plea for a sane versioning policy.



Major component

Increment the major component for all backwards incompatible changes.

Minor component

Increment the minor component for all backward compatible additions.

Micro component

Increment the micro component for all bug-fix releases.

1.3 Usage

The **versionah** script is the main workhorse of the *versionah* module.

Let's start with some basic examples:

```
$ versionah display _version.py  # Read the version data from _version.py
2.4.3
$ versionah bump _version.py minor  # Bump the minor component
2.5.0
$ versionah bump _version.py major  # Bump the major component
```

(continues on next page)

(continued from previous page)

```
3.0.0

$ versionah set _version.rb 0.2.0  # Set the version in _version.rb to 0.2.0
0.2.0

$ versionah bump _version.h minor  # Bump the minor component in _version.h
0.4.0
```

1.3.1 versionah

A tool to manage project version files.

```
versionah [OPTIONS] COMMAND [ARGS]...
```

Options

--version
Show the version and exit.

bump

Bump version in existing file.

```
versionah bump [OPTIONS] FILENAME... [major|minor|micro|patch]
```

Options

-d, --display <display_format>
Display format for output.

Options date|dict|dotted|hex|libtool|tuple|web

-t, --type <file_type>
Define the file type used for version file.

Options c|go|h|json|lua|m4|moon|nim|py|rb|text

--shtool, --no-shtool
Write shtool compatible output.

Arguments

FILENAME
Required argument(s)

BUMP
Required argument

display

Display version in given file.

```
versionah display [OPTIONS] FILENAME...
```

Options

-d, --display <display_format>
Display format for output.

Options date|dict|dotted|hex|libtool|tuple|web

Arguments

FILENAME
Required argument(s)

set

Set version in given file.

```
versionah set [OPTIONS] FILENAME... VERSION_STR
```

Options

-d, --display <display_format>
Display format for output.

Options date|dict|dotted|hex|libtool|tuple|web

-t, --type <file_type>
Define the file type used for version file.

Options c|go|h|json|lua|m4|moon|nim|py|rb|text

--shtool, --no-shtool
Write shtool compatible output.

-n, --name <name>
Package name for version(default from $PWD).

Arguments

FILENAME
Required argument(s)

VERSION_STR
Required argument

1.4 Getting started

1.4.1 Initial setup

The first time we run **versionah** we must supply the initial version number, and optionally a name for the package:

```
$ versionah set -t c src/version.h 0.2.0
0.2.0
$ versionah set -t c -n my_app src/version.h 0.2.0
0.2.0
```

1.4.2 `Makefile` usage

If your project uses make, it is a simple task to add version bumping rules:

```
$(addprefix version-, major minor micro):
    versionah bump src/version.h $(subst version-,, $@)
```

The above example makes it possible to call, for example, `version-minor` to bump the minor component in `src/version.h`.

Note: If you use automake then you can use the **PACKAGE_NAME** variable to set the *--name* value.

1.4.3 `libtool` example

It is easy to use the versioning information for libtool build rules in make files:

```
$(LIBRARY_NAME): $(LIBRARY_OBJS)
    $(LIBTOOL) --mode=link $(CC) -o $(LIBRARY_NAME) $(LIBRARY_OBJS) \
        -rpath $(libdir) \
        -version-info `versionah display -d libtool src/version.h`
```

Using the version information as the `libtool` interface age requires strict practise in maintaining the semantics of your version data, but doing so provides significant value to your users even if they aren't using the library interface.

1.4.4 `ninja` example

An example of usage from within ninja could be:

```
rule bump_versionah
    command = versionah bump example.txt $component
    description = BUMP $component

build version-major: bump_versionah
    component = major
build version-minor: bump_versionah
    component = minor
build version-micro: bump_versionah
    component = micro
```

Obviously, being a ninja you would choose to generate the `rule` and `build` directives programmatically.

1.4.5 `Sphinx` example

If you generate your project's documentation using Georg Brandl's excellent Sphinx tool, then you have a few options for including the version information.

Import the data

If you're storing your version data in Python format, then you can simply import the file. Accessing the data directly in your project's `conf.py`:

```
from versionah import _version
# The short X.Y version.
version = '{major}.{minor}'.format_map(jnrbase._version.dict)
# The full version
release = _version.dotted
```

Note: You may need to mangle `sys.path` if you can't import the version file from your `conf.py`.

Use the `versionah` output

Another option is to call **`versionah`** inside your `conf.py`:

```
import subprocess
# The full version
release = subprocess.check_output(["versionah", "versionah/_version.py"])
# The short X.Y version.
version = '.'.join(release.split('.')[:2])
```

The obvious drawback to this method is that it requires *all* users who wish to build the documentation to have **`versionah`** installed, and is therefore not recommended.

1.4.6 `pod2man` example

If you generate your documentation using perl's **`pod2man`**, then a sample `Makefile` rule to include your program's version information would be:

```
man.1: man.pod
    pod2man --section=1 \
        --release="`versionah display -d dotted src/version.h`" \
        --date="`versionah display -d date src/version.h`" $< $@
```

1.4.7 More examples

If you're using **`versionah`** with another common(-ish) tool, then new examples for this section are most welcome. Please consider posting them in an issue or pushing them to a fork on GitHub, so that others may benefit.

1.5 Version templates

Version files are created from templates using Armin Ronacher's excellent Jinja. Before writing your own templates you should read the splendid Jinja template designer documentation.

Note: If you create some cool templates of your own please consider posting them in an issue or pushing them to a fork on GitHub, so that others may benefit.

1.5.1 Template locations

Templates are loaded from directories in the following order:

- If it exists, `${XDG_DATA_HOME:~/.local/share}/versionah/templates`
- Any `versionah/templates` directory in the directories specified by *XDG_DATA_DIRS*
- The **versionah** package's `templates` directory

Note: For OS X users `versionah` will fallback to `~/Library/Application Support`, if *XDG_DATA_HOME* is unset.

1.5.2 Precedence

The first name match in the order *specified above* selects the template, so a `py.jinja` file in `$`*XDG_DATA_HOME*`/versionah/templates` overrides the `py.jinja` template provided with **versionah**.

1.5.3 Naming

Templates should be named after the common type suffix if possible, doing so allows **versionah** to guess an appropriate template from a supplied file. For example, `py.jinja` will apply by default to all files ending in `.py`.

Nevertheless, templates *can* be given any name you wish. This makes it simple to have project specific templates, should the need arise. This functionality is especially useful if you have shared data among a set of projects, as you can use Jinja's Template Inheritance support to reduce the duplication needed in each template.

1.5.4 Data

Each template is provided with the following variables for use in the output:

Variable	Content
`magic`	Magic string to support re-reading **versionah** files
`major` / `minor` / `micro` / `patch`	Individual component parts
`tuple`	Version components as a tuple of integers
`resolution`	The number of components used by the version object, mainly useful for advanced template logic
`name`	The package name
`dateobj`	Release date as a `datetime.date` object
`now`	File creation timestamp in the `local timezone`
`utcnow`	File creation timestamp in `UTC`
`filename`	Output file's name

In addition to the above list variables, all of the supported display methods[1] — for example `dotted` and `libtool` — are available for use too.

Jinja templates support object attribute and method access, so the `utcnow` object can be called with the `strftime()` method for custom timestamp output. For example, `{{ utcnow.strftime("%a, %e %b %Y %H:%M:%S %z") }}` could be used to output an **RFC 2822** date stamp[2].

The `text` display's template is simply:

```
{{ magic }}
```

which results in output such as:

```
This is mypkg version 2.2.4 (2011-02-19)
```

Note: If you're authoring your own templates and you find that you need extra data for use in their generation open an issue.

1.5.5 Filters

versionah defines various filters beyond the huge range of built-in filters in Jinja, please refer to the `jnrbase.template` documentation for more information.

Note: If you write extra filters and believe they could be of use to other **versionah** users please consider posting them in an issue or pushing them to a fork on GitHub, so that others may benefit from your work.

For example, the `regexp` filter is used in the C template to make valid identifiers from `filename` by replacing characters that are invalid in identifiers with underscores:

```
{% set escaped_name = filename|upper|regexp("[^A-Z]", "_") %}
```

[1] Technically, the result of any *Version* method beginning with `as_` is passed along to the template, with the `as_` prefixes removed.
[2] But don't do that, as `strftime()` is locale dependent.

1.6 versionah

1.6.1 Version management made easy

Author James Rowe <jnrowe@gmail.com>

Date 2011-02-15

Manual section 1

Manual group Developer

1.6.2 SYNOPSIS

versionah [OPTIONS] COMMAND [ARGS]...

1.6.3 DESCRIPTION

versionah is a simple tool to help you — or more specifically *me* — easily maintain version information for a project. Its entire aim is to make the act of displaying or bumping a project's version number a thoughtless task.

1.6.4 Options

--version
: Show the version and exit.

-h, --help
: Show this message and exit.

1.6.5 Commands

versionah bump

Bump version in existing file.

```
versionah bump [OPTIONS] FILENAME... [major|minor|micro|patch]
```

Options

-d, --display <display_format>
: Display format for output.

  Options date|dict|dotted|hex|libtool|tuple|web

-t, --type <file_type>
: Define the file type used for version file.

  Options c|go|h|json|lua|m4|moon|nim|py|rb|text

--shtool, --no-shtool
: Write shtool compatible output.

Arguments

FILENAME
Required argument(s)

BUMP
Required argument

versionah display

Display version in given file.

```
versionah display [OPTIONS] FILENAME...
```

Options

-d, --display <display_format>
Display format for output.

Options date|dict|dotted|hex|libtool|tuple|web

Arguments

FILENAME
Required argument(s)

versionah set

Set version in given file.

```
versionah set [OPTIONS] FILENAME... VERSION_STR
```

Options

-d, --display <display_format>
Display format for output.

Options date|dict|dotted|hex|libtool|tuple|web

-t, --type <file_type>
Define the file type used for version file.

Options c|go|h|json|lua|m4|moon|nim|py|rb|text

--shtool, --no-shtool
Write shtool compatible output.

-n, --name <name>
Package name for version(default from $PWD).

Arguments

FILENAME
Required argument(s)

VERSION_STR
Required argument

1.6.6 BUGS

None known.

1.6.7 AUTHOR

Written by James Rowe

1.6.8 RESOURCES

Documentation https://versionah.readthedocs.io/

Git repository https://github.com/JNRowe/versionah/

Issue tracker https://github.com/JNRowe/versionah/issues/

Contributors https://github.com/JNRowe/versionah/contributors/

1.6.9 COPYING

1.7 Frequently Asked Questions

1.7.1 Isn't this an overly elaborate solution for a simple problem?

Perhaps. However, it Works For Me™.

You're obviously free to use what works for you, be that manually updating the information in your editor or using a simpler approach such as shtool's `version` applet.

1.7.2 I give `perl` scripts the suffix `.perl`, can I make type guessing work?

Yes. The simplest way is to just create a symlink from **versionah**'s `pl.jinja`` to `${XDG_DATA_HOME:~/.local/share}/versionah/templates/perl.jinja`.

1.7.3 Do you accept template contributions?

Yes, if they are somewhat general.

Template contributions are also a great way to have me maintain template compatibility for you, and protects you from breaking changes in a future version.

Either open an issue or push them to a fork on GitHub.

1.7.4 I don't like your choice of template language

[It isn't really a question, but it has come up a couple of times.]

The use of Jinja should only be an issue if you wish to *author your own templates*, if you're using the built-in templates you shouldn't notice Jinja at all. That said...

The use of Jinja seems to be an unassailable barrier to entry for a couple of people, but it isn't going to change. For the same — invariably pointless and religious — reasons people prefer other templating engines *I* prefer Jinja.

1.7.5 How do I add version data to my project's `README`?

The way I manage it, using **versionah**, is by having a *custom template* for such a project.

The only templating requirement **versionah** has is that `{{ magic }}` is included *somewhere*. This means you can use a custom template that includes your full `README` data, and generate the distributed `README` from that. Consider it the `README.in` approach you've probably used with GNU autotools, and it makes perfect sense.

1.7.6 Will you support other version formats?

If the patches you submit for other version formats aren't too invasive then they'll probably be accepted. If you're going to propose such a patch open an issue or drop me a mail *first*, so it can be discussed.

If the format you're going to implement looks like the `LooseVersion` format defined in **PEP 386** with support for random words or odd characters then the answer is likely to be a resounding "no".

Direct support for the full `StrictVersion` format defined in that PEP will, however, likely be accepted with open arms.

1.8 Alternatives

Before diving in and spitting out this package I looked at the alternatives below. If I have missed something please drop me a mail.

1.8.1 shtool

shtool provides a great version management applet, one which I used for many years in various projects[1]. Unfortunately, the output formats are hard-coded in the script making it extremely difficult to use in the projects I work on.

A few ideas have been borrowed from `shtool`, and **versionah** should be seen as a homage to the version applet from `shtool`.

If you don't need the template support of **versionah** and find the other functionality `shtool` provides useful, then I'd strongly recommend using `shtool`. There is little point depending on two external projects when one can suffice.

Since version 0.8.0 of **versionah** it has been possible to parse `shtool` generated files[2]. You can also write `shtool`-compatible files with **versionah** v0.14.0 and later[3], see the *`versionah bump --shtool`* documentation.

1.9 Upgrading notes

Beyond the high-level notes you can find in *User-visible changes*, you'll find some more specific upgrading advice in this document.

1.10 User-visible changes

[1] According to the shtool ChangeLog I used it at least as far back as 2004 when I contributed M4 support, and I didn't spike **versionah** until 2011.

[2] 0.8.0 was the release I cut to as I started to migrate existing projects that used `shtool`.

[3] 0.14.0 was the release I cut to support a project that didn't want to migrate to **versionah**, where I had already become accustomed to **versionah**'s interface in the intervening three months.

1.10.1 0.16.0 - 2017-11-06

- Python 3 *only*, for Python 2 support you must use 0.15.0 or earlier
- Support for multiple output files per call
- `.` is now allowed in package names
- click and jnrbase `[colour,iso_8601,template]` are now required
- `aaargh` and `blessings` are no longer required
- pytest is used for running tests; `expecter`, `mock` and `nose` are no longer required

1.10.2 0.15.0 - 2014-01-31

- Defaults to bumping micro version, as that is the common case
- New templates for golang and lua

1.10.3 0.14.0 - 2013-05-22

- shtool compatible writing support

1.10.4 0.13.0 - 2013-05-09

- Switched to a subcommand-based interface
- aaargh is now required
- Support for localisation, submit pull requests with your translations!

1.10.5 0.12.0 - 2012-07-19

- blessings replaces termcolor for fancy output
- argparse is now required for Python 2.6
- Tests now use nose2, expecter and nose2-cov
- attest is no longer required for running tests
- cloud_sptheme is no longer required for building documentation
- On OS-X we fall back to `~/Library/Application Support` for templates

- pip requirements files are now included in `extra`

1.10.6 0.11.0 - 2012-01-30

- Added a basic JSON template, useful for handling version data for `nodejs` packages
- Improved developer documentation
- attest and behave are now required to run the testsuite

1.10.7 0.10.0 - 2011-03-30

- Added a `h` template, for writing C header files
- `dateobj` exported to templates for full access to datetime.date methods
- Man page available by calling `make man` in doc directory
- Sphinx examples in Getting Started document

1.10.8 0.9.0 - 2011-03-21

- Installable with setuptools
- Release date output with `--display date` option

1.10.9 0.8.0 - 2011-02-26

- Support for reading shtool generated files
- `major`, `minor`, `micro` and `patch` are exported to templates
- Added an M4 template, useful for working with autoconf

1.10.10 0.7.0 - 2011-02-21

- User templates now override system templates
- Ruby template
- Default file type is now based on version file's extension
- Support for comparing version numbers when used as a library

1.10.11 0.6.0 - 2011-02-19

- Support for versions containing two or four components
- `regexp` filter for use in custom templates

1.10.12 0.5.0 - 2011-02-19

- Includes a release date in version output
- Support for package names with dashes and underscores

1.10.13 0.4.0 - 2011-02-18

- Support for including a package name in output

1.10.14 0.3.0 - 2011-02-15

- Support for output templates using Jinja
- Coloured output using termcolor if available
- No longer supports Python 2.5 or lower

1.10.15 0.2.0 - 2011-02-15

- Python 3 support

1.10.16 0.1.0 - 2011-02-15

- Initial release

1.11 API documentation

Note: The documentation in this section is aimed at people wishing to contribute to *versionah*, and can be skipped if you are simply using the tool from the command line.

1.11.1 Command line

Note: The documentation in this section is aimed at people wishing to contribute to *versionah*, and can be skipped if you are simply using the tool from the command line.

class versionah.cmdline.**NameParamType**
Name parameter handler.

Initialise a new *ReMatchParamType* object.

class versionah.cmdline.**ReMatchParamType**
Regular expression based parameter matcher.

Initialise a new *ReMatchParamType* object.

convert(*value*, *param*, *ctx*)
Check given name is valid.

Parameters

- **value** (*str*) – Value given to flag
- **param** (*click.Argument*) – Parameter being processed
- **ctx** (*click.Context*) – Current command context

Returns Valid value

Return type str

class versionah.cmdline.**VersionParamType**
Version parameter handler.

Initialise a new *ReMatchParamType* object.

class versionah.cmdline.**CliVersion**(*components=(0, 1, 0)*, *name='unknown'*, *date=datetime.date(2019, 11, 7)*)
Specialisation of models.Version for command line usage.

Initialise a new *Version* object.

Parameters

- **components** (*str*) – Version components
- **name** (*str*) – Package name
- **date** (*datetime.date*) – Date associated with version

display(*display_format*)
Display a version string.

Parameters **display_format** (*str*) – Format to display version string in

Returns Formatted version string

Return type str

static display_types()
Supported representation types.

Returns Method names for representation types

Return type list[str]

static read(*filename*)
Read a version file.

Parameters **filename** (*str*) – Version file to read

Returns New *CliVersion* object representing file

Return type *CliVersion*

Raises

- OSError – When filename doesn't exist
- ValueError – Unparsable version data

write(*filename*, *file_type*, *, *shtool=False*)
Write a version file.

Parameters

- **filename** (*str*) – Version file to write
- **file_type** (*str*) – File type to write
- **shtool** (*bool*) – Write shtool compatible files

versionah.cmdline.**guess_type**(*filename*)
Guess output type from filename.

Parameters **filename** (*str*) – File to operate on

versionah.cmdline.**cli**(**args*, ***kwargs*)
Main command entry point.

versionah.cmdline.**bump**(**args*, ***kwargs*)
Bump version in existing file.

Parameters

- **display_format** (*str*) – Format to display output in
- **filename** (*tuple[str]*) – File to operate on
- **file_type** (*tuple[str]*) – File type to produce
- **shtool** (*bool*) – Write shtool compatible files
- **bump** (*str*) – Component to bump

versionah.cmdline.**display**(**args*, ***kwargs*)
Display version in given file.

Parameters

- **display_format** (*str*) – Format to display output in
- **filename** (*tuple[str]*) – File to operate on

versionah.cmdline.**set_version**(**args*, ***kwargs*)
Set version in given file.

Parameters

- **display_format** (*str*) – Format to display output in
- **filename** (*tuple[str]*) – File to operate on
- **file_type** (*tuple[str]*) – File type to produce
- **shtool** (*bool*) – Write shtool compatible files
- **name** (*str*) – Project name used in output
- **version_str** (*str*) – Initial version string

Examples

Reading version data from a file

```
>>> CliVersion.read('tests/data/test_a')
CliVersion((0, 1, 0), 'test', datetime.date(2011, 2, 19))
>>> CliVersion.read('tests/data/test_b')
CliVersion((1, 0, 0), 'test', datetime.date(2011, 2, 19))
>>> CliVersion.read('tests/data/test_c')
CliVersion((2, 1, 3), 'test', datetime.date(2011, 2, 19))
```

Writing version date to a file

```
>>> v = CliVersion((0, 1, 0), 'test', datetime.date(2011, 2, 19))
>>> v.write('test_data.python', 'py')  # doctest: +SKIP
>>> v.write('test_data.hh', 'h')  # doctest: +SKIP
>>> v.write('test_data.m4', 'm4')  # doctest: +SKIP
```

Guess file type from name

```
>>> guess_type('main.c')
'c'
>>> guess_type('version.py')
'py'
>>> guess_type('no_suffix')
'text'
>>> guess_type('suffix.unknown')
'text'
```

1.11.2 Version

Note: The documentation in this section is aimed at people wishing to contribute to *versionah*, and can be skipped if you are simply using the tool from the command line.

versionah.models.**VALID_PACKAGE = '[A-Za-z][A-Za-z0-9]+(?:[_\\.-][A-Za-z0-9]+)*'**
Regular expression to match a valid package name

versionah.models.**VALID_VERSION = '\\d+\\.\\d+(?:\\.\\d+){,2}'**
Regular expression to match a valid package version

versionah.models.**VALID_DATE = '(?:\\d{4}-\\d{2}-\\d{2}|\\d{2}-(?:Jan|Feb|Mar|Apr|May|Jun|Ju**
Regular expression to match a package date. ISO-8601, and %d-%b-%Y formatting for shtool compatibility

versionah.models.**VERSION_COMPS = ('major', 'minor', 'micro', 'patch')**
Supported version components

class versionah.models.**Version**(*components=(0, 1, 0), name='unknown', date=datetime.today()*)
Main version identifier representation.

Initialise a new *Version* object.

Parameters

- **components** (*str*) – Version components
- **name** (*str*) – Package name
- **date** (*datetime.date*) – Date associated with version

as_date()
Generate a ISO-8601 date string for release.

Returns Version's release date as ISO-8601 date stamp

Return type str

as_dict()
Generate a dictionary of version components.

Returns Version as dictionary

Return type dict

as_dotted()
Generate a dotted version string.

Returns Standard dotted version string

Return type str

as_hex()
Generate a hex version string.

Returns Version as hex string

Return type str

as_libtool()
Generate a libtool version string.

Returns Version as libtool string

Return type str

as_tuple()
Generate a tuple of version components.

Returns Version components as tuple

Return type int

as_web()
Generate a web UA-style string for release.

Returns Version's string in web UA-style

Return type str

bump(*bump_type*)
Bump a version string.

Parameters **bump_type** (*str*) – Component to bump

Raises ValueError – Invalid `bump_type` argument

bump_major()
Bump major version component.

bump_micro()
Bump micro version component.

bump_minor()
Bump minor version component.

bump_patch()
Bump patch version component.

components
Generate component tuple to initial resolution.

Returns tuple[int]

components_full
Generate full length component tuple for version.

Returns tuple[int]

set(*components*)
Set version components.

Parameters **components** (*tuple[int]*) – Version components

versionah.models.**split_version**(*version*)
Split version string to components.

Parameters **version** (*str*) – Version string

Returns Components of version string

Return type tuple[int]

Raises ValueError – Invalid version string

Examples

Bumping a version component

```
>>> v = Version((0, 1, 0), 'test', datetime.date(2011, 2, 19))
>>> v.bump('minor')
>>> v.components
(0, 2, 0)
>>> v.bump('major')
>>> v.components
(1, 0, 0)
>>> v.bump_major()
>>> v.components
(2, 0, 0)
```

Version string parsing

```
>>> split_version('4.3.0')
(4, 3, 0)
>>> split_version('4.3.0.1')
(4, 3, 0, 1)
>>> split_version('4.3.0.1.3')
Traceback (most recent call last):
    ...
ValueError: Invalid version string '4.3.0.1.3'
```

1.12 Glossary

1.13 Release HOWTO

1.13.1 Test

Tests can be run via pytest:

```
$ pip install -r extra/requirements-test.txt
$ pytest -v tests
```

When preparing a release it is important to check that **versionah** works with all supported Python versions, and that the documentation for executing them is correct.

1.13.2 Prepare release

With the tests passing, do the following steps:

- Update the version data in `versionah/_version.py`
- Update `NEWS.rst` with any user visible changes
- Commit the release notes and version changes
- Create a signed tag for the release
- Push the changes — including the new tag — to the GitHub repository
- Create a new release on GitHub

1.13.3 Update PyPI (Python Package Index)

Create and upload the new release tarballs to PyPI using twine:

```
$ ./setup.py sdist bdist_wheel
$ gpg --detach-sign --armour dist/versionah-${version}.tar.gz
$ gpg --detach-sign --armour dist/versionah-${version}-*.whl
$ twine upload dist/versionah-${version}*
```

Fetch the uploaded tarballs, and check for errors.

You should also test installation from PyPI, to check the experience **versionah**'s end users will have.

1.14 Appendix

XDG_DATA_DIRS
: For information on the usage of `XDG_DATA_DIRS` read XDG Base Directory Specification

XDG_DATA_HOME
: For information on the usage of `XDG_DATA_HOME` read XDG Base Directory Specification

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

N

P

R

S

V

W

X