Releases

Release

Dec 31, 2017

Contents

1 What is Releases? 3

2 Table of Contents 5
- 2.1 Concepts 5
- 2.2 Usage 14
- 2.3 Changelog 16


build
passing

CHAPTER 1

What is Releases?

Releases is a Python (2.7, 3.4+) compatible Sphinx (1.3+) extension designed to help you keep a source control friendly, merge friendly changelog file & turn it into useful, human readable HTML output.

Specifically:

- The source format (kept in your Sphinx tree as `changelog.rst`) is a stream-like timeline that plays well with source control & only requires one entry per change (even for changes that exist in multiple release lines).
- The output (when you have the extension installed and run your Sphinx build command) is a traditional looking changelog page with a section for every release; multi-release issues are copied automatically into each release.
- By default, feature and support issues are only displayed under feature releases, and bugs are only displayed under bugfix releases. This can be overridden on a per-issue basis.

Some background on why this tool was created can be found in this blog post.

For more documentation, please see http://releases.readthedocs.io.

Note: You can install the development version via `pip install -e git+https://github.com/bitprophet/releases/#egg=releases`.

CHAPTER 2

Table of Contents

2.1 Concepts

This page contains conceptual info about how Releases organizes and thinks about issues and releases. For details on formatting/config options/etc (e.g. so you can interpret the examples below), see *Usage*.

2.1.1 Issue and release types

- Issues are always one of three types: **features**, **bug fixes** or **support items**.
  - **Features** are (typically larger) changes adding new behavior.
  - **Bug fixes** are (typically minor) changes addressing incorrect behavior, crashes, etc.
  - **Support items** vary in size but are usually non-code-related changes, such as documentation or packaging updates.
- Releases also happen to come in three flavors:
  - **Major releases** are backwards incompatible releases, often with large/sweeping changes to a codebase.
    - They increment the first version number only, e.g. `1.0.0`.
  - **Feature releases** (sometimes called **minor** or **secondary**) are backwards compatible with the previous major release, and focus on adding new functionality (code, or support, or both.) They sometimes include major/complex bug fixes which are too risky to include in a bugfix release.
    - The second version number is incremented for these, e.g. `1.1.0`.
  - **Bugfix releases** (sometimes called **tertiary**) focus on fixing incorrect behavior while minimizing the risk of creating more bugs. Rarely, they will include small new features deemed important enough to backport from their 'native' feature release.
    - These releases increment the third/final version number, e.g. `1.1.1`.

2.1.2 Release organization

We parse changelog timelines so the resulting per-release issue lists honor the above descriptions. Here are the core rules, with examples. See *Usage* for details on formatting/etc.

- **By default, bugfixes go into bugfix releases, features and support items go into feature releases.**
  - Input:

```
* :release:`1.1.0 <date>`
* :release:`1.0.1 <date>`
* :support:`4` Updated our test runner
* :bug:`3` Another bugfix
* :feature:`2` Implemented new feature
* :bug:`1` Fixed a bug
* :release:`1.0.0 <date>`
```

  - Result:
    - `1.1.0`: feature #2, support #4
    - `1.0.1`: bug #1, bug #3
- **Bugfixes are assumed to backport to all stable release lines by default, and are displayed as such.** However, this can be overridden on a per-release and/or per-bug basis - see later bullet points.
  - Input:

```
* :release:`1.1.1 <date>`
* :release:`1.0.2 <date>`
* :bug:`3` Fixed another bug, onoes
* :release:`1.1.0 <date>`
* :release:`1.0.1 <date>`
* :feature:`2` Implemented new feature
* :bug:`1` Fixed a bug
* :release:`1.0.0 <date>`
```

  - Result:
    - `1.1.1`: bug #3
    - `1.0.2`: bug #3
    - `1.1.0`: feature #2
    - `1.0.1`: bug #1
- **Bugfixes marked 'major' go into feature releases instead.** In other words, they're displayed as bugs, but organized as features.
  - Input:

```
* :release:`1.1.0 <date>`
* :release:`1.0.1 <date>`
* :bug:`3 major` Big bugfix with lots of changes
* :feature:`2` Implemented new feature
* :bug:`1` Fixed a bug
* :release:`1.0.0 <date>`
```

  - Result:
    - `1.1.0`: feature #2, bug #3

    * `1.0.1`: bug #1

- **Features or support items marked 'backported' appear in both bugfix and feature releases.** In other words, they're displayed as feature/support items, but organized as a combination feature/support *and* bug item.
  - Input:

```
* :release:`1.1.0 <date>`
* :release:`1.0.1 <date>`
* :bug:`4` Fixed another bug
* :feature:`3` Regular feature
* :feature:`2 backported` Small new feature worth backporting
* :bug:`1` Fixed a bug
* :release:`1.0.0 <date>`
```

  - Result:

    * `1.1.0`: feature #2, feature #3

    * `1.0.1`: bug #1, feature #2, bug #4

- **Releases implicitly include all issues from their own, and prior, release lines.** (Again, unless the release explicitly states otherwise - see below.)
  - For example, in the below changelog (remembering that changelogs are written in descending order from newest to oldest entry) the code released as `1.1.0` includes the changes from bugs #1 and #3, in addition to its explicitly stated contents of feature #2:

```
* :release:`1.1.0 <date>`
* :release:`1.0.1 <date>`
* :bug:`3` Another bugfix
* :feature:`2` Implemented new feature
* :bug:`1` Fixed a bug
* :release:`1.0.0 <date>`
```

  - Again, to be explicit, the rendered changelog displays this breakdown:

    * `1.1.0`: feature #2

    * `1.0.1`: bug #1, bug #3

    But it's *implied* that `1.1.0` includes the contents of `1.0.1` because it released afterwards/simultaneously and is a higher release line.

- **Releases may be told explicitly which issues to include** (using a comma-separated list.) This is useful for the rare bugfix that gets backported beyond the actively supported release lines.

  For example, below shows a project whose lifecycle is "release 1.0; release 1.1 and drop active support for 1.0; put out a special 1.0.x release." Without the explicit issue list for 1.0.1, Releases would roll up all bugfixes, including the two that didn't actually apply to the 1.0 line.

  - Input:

```
* :release:`1.0.1 <date>` 1, 5
* :release:`1.1.1 <date>`
* :bug:`5` Bugfix that applied back to 1.0.
* :bug:`4` Bugfix that didn't apply to 1.0, only 1.1
* :bug:`3` Bugfix that didn't apply to 1.0, only 1.1
* :release:`1.1.0 <date>`
* :feature:`2` Implemented new feature
* :bug:`1` Fixed a 1.0.0 bug
* :release:`1.0.0 <date>`
```

  - Result:
    - `1.1.0`: feature #2
    - `1.1.1`: bugs #3, #4 and #5
    - `1.0.1`: bugs #1 and #5 only

- **Bugfix issues may be told explicitly which release line they 'start' in.** This is useful for bugs that don't go back all the way to the oldest actively supported line - it keeps them from showing up in "too-old" releases.

  The below example includes a project actively supporting 1.5, 1.6 and 1.7 release lines, with a couple of bugfixes that only applied to 1.6+.

  - Input:

```
* :release:`1.7.1 <date>`
* :release:`1.6.2 <date>`
* :release:`1.5.3 <date>`
* :bug:`50` Bug applying to all lines
* :bug:`42 (1.6+)` A bug only applying to the new feature in 1.6
* :release:`1.7.0 <date>`
* :release:`1.6.1 <date>`
* :release:`1.5.2 <date>`
* :feature:`25` Another new feature
* :bug:`35` Bug that applies to all lines
* :bug:`34` Bug that applies to all lines
* :release:`1.6.0 <date>`
* :release:`1.5.1 <date>`
* :feature:`22` Some new feature
* :bug:`20` Bugfix
* :release:`1.5.0 <date>`
```

  - Result:
    - `1.7.1`: bugs #50 and #42
    - `1.6.2`: bugs #50 and #42
    - `1.5.3`: bug #50 only
    - `1.7.0`: feature #25
    - `1.6.1`: bugs #34, #35
    - `1.5.2`: bugs #34, #35
    - `1.6.0`: feature #22
    - `1.5.1`: bug #20

- **Bugs listed before the first release are treated as though they have the 'major' keyword.** This is chiefly because it makes no sense to have a "bugfix release" as one's first-ever release - you can't fix something that's not public! Then once the changelog parser passes that initial release, normal rules start to apply again.

  - Input:

```
* :release:`0.1.1`
* :bug:`3` The feature had bugs :(
* :release:`0.1.0 <date>`
* :feature:`2` Our first ever feature
* :bug:`1` Explicitly marked bug, even though that is silly
* Implicit issue/entry here (becomes a bug by default)
```

  - Result:
    * `0.1.1`: bug #3 only, since it's the only bug after the first release.
    * `0.1.0`: everything else - the implicit bug, the explicit bug #1, and the feature #2.

2.1.3 Major releases

Major releases introduce additional concerns to changelog organization on top of those above. Users whose software tends to just "roll forwards" without keeping older stable branches alive for bugfix releases, will likely not need to do much.

However, when your support window stretches across major version boundaries, telling Releases which issues belong to which major version (or versions plural) becomes a bit more work.

There are two main rules to keep in mind when dealing with "mixed" major versions:

- **All issues encountered after a major release** are considered associated with that major release line **by default**.
- **All feature-like items (features, support, major bugs) encountered just prior to a major release** are considered part of the major release itself.
- To force association with a **different major release** (or set of major releases), issues may **specify a 'version spec'** annotation.

Here's some examples to clarify.

"Rolling" releases

This example has no mixing of release lines, just moving from 1.x to 2.x. 1.x is effectively abandoned. (Hope 2.x is an easy upgrade...) Note how features 4 and 5, because they are encountered prior to 2.0.0, are attached to it automatically.

Input:

```
* :release:`2.1.0 <date>`
* :release:`2.0.1 <date>`
* :feature:`7` Yet another new feature
* :bug:`6` A bug :(
* :release:`2.0.0 <date>`
* :feature:`5` Another (backwards incompatible) feature!
* :feature:`4` A (backwards incompatible) feature!
* :release:`1.1.0 <date>`
* :release:`1.0.1 <date>`
* :feature:`3` New feature
* :bug:`2` Another bug
* :bug:`1` An bug
* :release:`1.0.0 <date>`
```

Result:

- `2.1.0`: feature #7
- `2.0.1`: bug #6
- `2.0.0`: feature #4, feature #5
- `1.1.0`: feature #3
- `1.0.1`: bug #1, bug #2

Pretty simple, nothing actually new here.

Mostly-compatible 2.0 with continued maint for 1.x

This maintainer is a bit more conscientious/masochistic and wants to keep users of 1.x happy for a while after 2.0 launches.

The timeline is very similar to the previous example, but in this scenario, all issues developed on the 1.x branch are forward-ported to 2.x, because 2.x wasn't a huge departure from 1.x.

To signify this, post-2.0 issues that were developed initially for 1.x, are annotated with `(1.0+)`, telling Releases to add them to all releases above 1.0, instead of just the most recent major release (2.0):

```
* :release:`2.1.0 <date>`
* :release:`2.0.1 <date>`
* :release:`1.2.0 <date>`
* :release:`1.1.1 <date>`
* :release:`1.0.2 <date>`
* :bug:`9` A 2.0-only bugfix.
* :feature:`8` A 2.0-only feature.
* :feature:`7 (1.0+)` Yet another new feature
* :bug:`6 (1.0+)` A bug :(
* :release:`2.0.0 <date>`
* :feature:`5` Another (backwards incompatible) feature!
* :feature:`4` A (backwards incompatible) feature!
* :release:`1.1.0 <date>`
* :release:`1.0.1 <date>`
* :feature:`3` New feature
* :bug:`2` Another bug
* :bug:`1` An bug
* :release:`1.0.0 <date>`
```

Result:

- `2.1.0`: feature #7, feature #8
- `2.0.1`: bug #6, bug #9
- `1.2.0`: feature #7, but not feature #8
- `1.1.1`: bug #6, but not bug #9
- `1.0.2`: bug #6, but not bug #9
- `2.0.0`: feature #4, feature #5
- `1.1.0`: feature #3
- `1.0.1`: bug #1, bug #2

Some issues forward-ported, others not

This time, some issues remain 1.x-specific as they don't apply to 2.x for whatever reason. The simple "X.Y+" format doesn't let us declare this, so we use one you're familiar with from packaging systems like `setuptools`/`pip`:

- `(<2.0)` signifies "only included in releases lower than 2.0"
- `(>=2.0)` says "only include in release lines 2.0 and higher" (thus applying to 2.1, 2.2, 3.0, 4.0 etc).
  - This is identical to saying `(2.0+)`; the `+` version is just a convenient / backwards compatible shorthand.

- `(>=2.0,<3.0)` limits an issue to *just* the 2.x line, preventing its inclusion in 1.x, 3.x or anything else.
- And so on; see the documentation for the `Spec` class at https://python-semanticversion.readthedocs.io for details.
- To be clear, **you may put any combination of major+minor version number in these annotations**, just as with the simpler `(1.5+)` style format.
  - This is mostly applicable to bugs or backported issues. Features, support items and major bugs only need to inform Releases about major release lines.

Armed with this more powerful syntax, we can limit some issues just to the 1.x line:

```
* :release:`2.1.0 <date>`
* :release:`2.0.1 <date>`
* :release:`1.2.0 <date>`
* :release:`1.1.1 <date>`
* :release:`1.0.2 <date>`
* :feature:`9 (>=1.0)` A new feature that works with both versions (using
  the more explicit version of "1.0+")
* :feature:`8` A new feature that only works on 2.x (no annotation needed)
* :bug:`7 (<2.0)` A bug only affecting 1.x
* :bug:`6 (1.0+)` A bug affecting all versions
* :release:`2.0.0 <date>`
* :feature:`5` Another (backwards incompatible) feature!
* :feature:`4` A (backwards incompatible) feature!
* :release:`1.1.0 <date>`
* :release:`1.0.1 <date>`
* :feature:`3` New feature
* :bug:`2` Another bug
* :bug:`1` An bug
* :release:`1.0.0 <date>`
```

Result:

- `2.1.0`: feature #8, feature #9
- `2.0.1`: bug #6 (but not #7)
- `1.2.0`: feature #9 (but not #8)
- `1.1.1`: bug #6, bug #7
- `1.0.2`: bug #6, bug #7
- `2.0.0`: feature #4, feature #5
- `1.1.0`: feature #3
- `1.0.1`: bug #1, bug #2

Mixed-but-exclusive features prior to a new major release

This example illustrates a corner case where one is actively maintaining a "current" 1.x line at the same time as releasing the new 2.x line. Unlike the earlier examples, this one has both "2.0-only" *and* "1.0-only" features in the run-up to 2.0.0 (plus bugs).

In this scenario, the non-annotated features are automatically assigned to the 2.0 major version, even though the 1.2.0 release technically came out "before" 2.0.0.

As long as no non-release line items appear between 1.2.0 and 2.0.0, the system will behave as if 2.0.0 was the "primary" next release, with 1.2.0 only capturing features explicitly annotated as being "<2.0" (or similar).

Note: This behavior holds true even if the adjacent release line-items have different dates; the heuristic is solely about their placement in the changelog list.

Note also how bugs found in this window just prior to 2.0.0, remain associated with the 1.x line that they are fixing; it wouldn't make sense to publish a bugfix for unreleased functionality.

Changelog:

```
* :release:`2.0.0 <date>`
* :release:`1.2.0 <date>`
* :release:`1.1.1 <date>`
* :bug:`6` A bug found after 1.1.0 came out
* :feature:`5 (<2.0)` A 1.0-only feature!
* :feature:`4` A (backwards incompatible) feature!
* :release:`1.1.0 <date>`
* :release:`1.0.1 <date>`
* :feature:`3` New feature
* :bug:`2` Another bug
* :bug:`1` An bug
* :release:`1.0.0 <date>`
```

Result:

- `2.0.0`: feature #4 (but not feature #5)
- `1.2.0`: feature #5 (but not feature #4)
- `1.1.1`: bug 6
- `1.1.0`: feature #3
- `1.0.1`: bug #1, bug #2

2.1.4 "Unstable prehistory" mode

All of the above assumes a mature, semantic-versioning-enabled project, where you have stable release lines as well as a feature development 'trunk' branch. This doesn't always describe young projects, however - before one's 1.0.0, semantic versioning may not apply strongly or at all.

When the `releases_unstable_prehistory` option is enabled (it's off by default for backwards compatibility reasons), changelog parsing/organizing behaves differently, until releases other than `0.x.x` are encountered:

- All issues, regardless of type, are assigned to the very next release; there's no organizing along minor release lines, no 'major' bugs are necessary, nor are 'backported' features.
- Unmarked line-items - which are normally considered to be bugs - are displayed without any classification (i.e. they don't get a 'Bug' prefix).
  - This is mostly to enable the types of "pre-Releases" changelogs wherein *all* line items lack issue-type role prefixes.
  - If your changelog *does* include explicit role prefixes (`:bug:`, `:feature:` etc) they are left untouched & will still visually appear as the indicated type.

Example

Here's an example of what this option means. Take the following changelog:

```
* :release:`0.2.1 <date>`
* Bugfix #7
* Feature #6, but meh, we arbitrarily are gonna call the next release a
  tertiary one anyways
* Bugfix #5
* :release:`0.2.0 <date>`
* Medium bugfix #4
* Tiny bugfix #3
* Feature #2
* :release:`0.1.0 <date>`
* It works! First public release.
```

Under normal Releases behavior this wouldn't match what the author clearly intends - all of these line items lack roles, so they'd all be "bugs", and then none of them would get inserted into 0.1.0 or 0.2.0 which are feature releases.

With `releases_unstable_prehistory` enabled, we instead get:

- `0.2.1`: bugfix 5, feature 6, bugfix 7
- `0.2.0`: feature 2, bugfix 3, bugfix 4
- `0.1.0`: the beginning-of-time "it works!" note

Crossing the 1.0 boundary

As mentioned, even when this option is enabled, the 1.0.0 release (or whichever release is the first not beginning with `0.`) implicitly deactivates this behavior. All subsequent issues then follow the behavior outlined in the rest of the document: bugfixes only go in tertiary releases, features only go in minor releases, etc.

Another explicit example - this changelog (which is even more arbitrary with its versioning prior to 1.0):

```
* :release:`1.1.0 <date>`
* :release:`1.0.1 <date>`
* :feature:`8` A new, backwards compatible feature, hooray
* :bug:`7` First post-1.0 bugfix!
* :release:`1.0.0 <date>`
* Bug #6
* Feature #5
* `0.5.0`
* Feature #4
* Bug #3
* Bug #2
* `0.1.0`
* Feature #1
```

The resulting changelog is organized like so:

- `1.1.0`: Feature #8
- `1.0.1`: Bug #7 - no features, this is the first "real" bugfix release
- `1.0.0`: Bug #6, feature #5 - this is the last "unstable" release rolling up all prior issues.
- `0.5.0`: Bug #2, bug #3, feature #4
- `0.1.0`: Feature #1

2.2 Usage

To use Releases, mimic the format seen in its own changelog or in Fabric's changelog. Specifically:

- Install `releases` and update your Sphinx `conf.py` to include it in the `extensions` list setting: `extensions = ['releases']`.
  - Also set the `releases_release_uri` and `releases_issue_uri` top level options - they determine the targets of the issue & release links in the HTML output. Both should have an unevaluated `%s` where the release/issue number would go.
    - Alternately, if your project is hosted on Github, set the `releases_github_path` setting instead, to e.g. `account/project`. Releases will then use an appropriate Github URL for both releases and issues.
    - If `releases_release_uri` or `releases_issue_uri` are *also* configured, they will be preferred over `releases_github_path`. (If only one is configured, the other link type will continue using `releases_github_path`.)
  - See Fabric's docs/conf.py for an example.
  - You may optionally set `releases_debug = True` to see debug output while building your docs.
  - If your changelog includes "simple" pre-1.0 releases derived from a single branch (i.e. without stable release lines & semantic versioning) you may want to set `releases_unstable_prehistory = True`.
    - This is also useful if you've just imported a non-Releases changelog, where your issues are all basic list-items and you don't want to go through and add bug/feature/support/etc roles.
    - See *the appropriate conceptual docs* for details on this behavior.
- Create a Sphinx document named `changelog.rst` containing a bulleted list somewhere at its topmost level.
  - If you wish to use a different document name, use another config option (as per previous bullet point), `releases_document_name`. E.g. `releases_document_name = "CHANGES"` would cause Releases to mutate a file called `CHANGES.rst` instead of `changelog.rst`.
  - Elements before or after this bulleted list will be untouched by Releases, allowing you to place e.g. paragraphs, comments etc at the top (or bottom) of the document.
- List items are to be ordered chronologically with the newest ones on top.
  - As you fix issues, put them on the top of the list.
  - As you cut releases, put those on the top of the list and they will include the issues below them.
  - Issues with no releases above them will end up in a specially marked "Unreleased" section of the rendered changelog.
- Bullet list items should use the `support`, `feature` or `bug` roles to mark issues, or `release` to mark a release. These special roles must be the first element in each list item.
  - Line-items that do not start with any issue role will be considered bugs (both in terms of inclusion in releases, and formatting) and, naturally, will not be given a hyperlink.
- Issue roles are of the form :type:`number[ keyword]`. Specifically:
  - `number` is used to generate the link to the actual issue in your issue tracker (going by the `releases_issue_uri` option). It's used for both the link target & (part of) the link text.
  - If `number` is given as `-` or `0` (as opposed to a "real" issue number), no issue link will be generated. You can use this for items without a related issue.

  - Keywords are optional and may be one of:
    * `backported`: Given on *support* or *feature* issues to denote backporting to bugfix releases; such issues will show up in both release types. E.g. placing `:support:`123 backported`` in your changelog below releases '1.1.1' and '1.2.0' will cause it to appear in both of those releases' lists.
    * `major`: Given on *bug* issues to denote inclusion in feature, instead of bugfix, releases. E.g. placing `:bug:`22 major`` below releases '1.1.1' and '1.2.0' will cause it to appear in '1.2.0' **only**.
    * `(N.N+)` where `N.N` is a valid release line, e.g. `1.1` or `2.10`: Given on issues (usually *bugs*) to denote minimum release line. E.g. when actively backporting most bugs to release lines 1.2, 1.3 and 1.4, you might specify `:bug:`55 (1.3+)`` to note that bug 55 only applies to releases in 1.3 and above - not 1.2.
    * A semantic version range spec covering minor+major version numbers such as `(<2.0)` or `(>=1.0,<3.1)`. A more powerful version of `(N.N+)` allowing annotation of issues belonging to specific major versions.

    Note: It is possible to give *both* a regular keyword (`backported`/`major`) *and* a spec (`(N.N+)`/`(>=1.0)`) in the same issue. However, giving two keywords or two specs at the same time makes no sense & is not allowed.

- Regular Sphinx content may be given after issue roles and will be preserved as-is when rendering. For example, in `:bug:`123` Fixed a bug, thanks `@somebody`!`, the rendered changelog will preserve/render "Fixed a bug, thanks `@somebody`!" after the issue link.
- Release roles are of the form `:release:`number <date>``.
  - You may place a comma-separated (whitespace optional) list of issue numbers after the release role, and this will limit the issues included in that release to that explicit list.
  - Otherwise, releases include all relevant issues as outlined above and in *Concepts*.

Then build your docs; in the rendered output, `changelog.html` should show issues grouped by release, as per the above rules. Examples: Releases' own rendered changelog, Fabric's rendered changelog.

2.2.1 Optional styling additions

If you have any nontrivial changelog entries (e.g. whose description spans multiple paragraphs or includes their own bulleted lists, etc) you may run into docutils' rather enthusiastic bulleted list massaging which can then make your releases look different from one another.

To help combat this, it may be useful to add the following rule to the Sphinx theme you're using:

```
div#changelog > div.section > ul > li > p:only-child {
    margin-bottom: 0;
}
```

Note: Some themes, like Alabaster, may already include this style rule.

2.3 Changelog

- : Identified a handful of issues with our Sphinx pin & subsequently, internal changes in Sphinx 1.6 which broke (and/or appear to break, such as noisy warnings) our own behavior. These have (hopefully) all been fixed.
- #68: Update packaging requirements to allow for `sphinx>=1.3,<2`. Thanks to William Minchin.
- : Drop Python 2.6 and 3.3 support, to correspond with earlier changes in Sphinx and most other public Python projects.
- #68: Update packaging requirements to allow for `sphinx>=1.3,<2`. Thanks to William Minchin.
- #66: (via #67) Deal with some Sphinx 1.6.1 brokenness causing `AttributeError` by leveraging `getattr()`'s default-value argument. Thanks to Ian Cordasco for catch & patch.
- #60: Report extension version to Sphinx for improved Sphinx debug output. Credit: William Minchin.
- : Add `releases.util`, exposing (among other things) a highly useful `parse_changelog(path)` function that returns a user-facing dict representing a parsed changelog. Allows users to examine their changelogs programmatically and answer questions like "do I have any outstanding bugs in the 1.1 release line?".
- #51: Modernize release management so PyPI trove classifiers are more accurate, wheel archives are universal instead of Python 2 only, and release artifacts are GPG signed.
- #36: Changelogs with no releases whatsoever should still be viable instead of raising exceptions. This is now happily the case. All items in such changelogs will end up in a single "unreleased features" list, just as with regular prehistory entries. Thanks to Steve Ivy for initial report and André Caron for additional feedback.
- #56: Fix exceptions that occurred when no release/issue link options were configured. Now those options are truly optional: release version and issue number text will simply display normally instead of as hyperlinks. Thanks to André Caron for the report.
- #51: Modernize release management so PyPI trove classifiers are more accurate, wheel archives are universal instead of Python 2 only, and release artifacts are GPG signed.
- #19: Add `unstable_prehistory` option/mode for changelogs whose 0.x release cycle is "rapid" or "unstable" and doesn't closely follow normal semantic version-driven organization. See *"Unstable prehistory" mode*.
- #53: Tweak newly-updated models so bugfix items prior to an initial release are considered 'major bugs' so they get rolled into that initial release (instead of causing a `ValueError`).
- #55: Non-annotated changelog line items (which implicitly become bugs) were incorrectly truncating their contents in some situations (basically, any time they included non-regular-text elements like monospace, bold etc). This has been fixed.
- : Fix formatting of release header dates; a "75% text size" style rule has had an uncaught typo for some time.
- #45: Add support for major version transitions (e.g. 1.0 to 2.0).

  Note: This adds a new install-time dependency: the semantic_version library. It's pure Python, so installation should be trivial.

- #44: Update one of our internal docutils-related classes for compatibility with Sphinx 1.4.x. Thanks to Gabi Davar for catch & patch.
- #41: Clean up changelog discovery so one can have comments, paragraphs or other non-bullet-list elements above or below the changelog. Thanks to Rodrigue Cloutier for the original request/patch.
- #42: For readability, issues within each release so they are displayed in feature->bug->support order.

- #21: Allow duplicate issue numbers; not allowing them was technically an implementation detail. Thanks to Dorian Puła for the patch.
- #30: Add LICENSE (plus a handful of other administrative files) to a `MANIFEST.in` so sdists pick it up. Thanks to Zygmunt Krynicki for catch & original patch (#33).
- : Fix a silly issue with the new feature from #22 where it accidentally referred to the Sphinx document *title* instead of the document *filename*.
- #26: Allow specifying Github path shorthand config option instead of explicit release/issue URL strings.
- #22: Make the document name used as the changelog - previously hardcoded as `changelog` (`.rst`) - configurable. Thanks to James Mills for the feature request.
- #24: Broke inline issue parsing; fixed now.
- #25: Empty/no-issue line items broke at some point; fixed.
- #23: Rework implementation to deal with issue descriptions that span more than one paragraph - subsequent paragraphs/blocks were not being displayed prior.
- : Fix silly bug in #20 that cropped up on Python 3.x.
- #20: Allow specifying minimum release line in bugfixes that don't apply to all active lines (e.g. because they pertain to a recently added feature.)
- #17: Allow releases to explicitly define which issues they include. Useful for overriding default assumptions (e.g. a special bugfix release from an otherwise dormant line.)
- #15: Add *Concepts* to flesh out some assumptions not adequately explained in *Usage*.
- #16: Fix some edge cases regarding release ordering & unreleased issue display. Includes splitting unreleased display info into two 'Next release' pseudo-release entries.
- #1: (also #3, #10) Allow using `-` or `0` as a dummy issue 'number', which will result in no issue number/link being displayed. Thanks to Markus Zapke-Gründemann and Hynek Schlawack for patches & discussion.
  - This feature lets you categorize changes that aren't directly related to issues in your tracker. It's an improvement over, and replacement for, the previous "vanilla bullet list items are treated as bugs" behavior.
  - Said behavior (non-role-prefixed bullet list items turning into regular bugs) is being retained as there's not a lot to gain from deactivating it.
- #11: Fix up styling so changelogs don't look suboptimal under the new Read The Docs theme. Still looks OK under their old theme too!
- #9: Clean up additional 'unreleased' display/organization behavior, including making sure ALL unreleased issues show up as 'unreleased'. Thanks to Donald Stufft for the report.
- : Explicitly documented how non-role-prefixed line items are preserved.
- : Edited the README/docs to be clearer about how Releases works/operates.
- : Handful of typos, doc tweaks & addition of a .gitignore file. Thanks to Markus Zapke-Gründemann.
- : Created a basic test suite to protect against regressions.
- : Move to actual Sphinx docs so we can use ourselves.
- : Updated non-role-prefixed line items so they get prefixed with a '[Bug]' signifier (since they are otherwise treated as bugfix items.)
- : Fix duplicate display of "bare" (not prefixed with an issue role) changelog entries. Thanks again to Markus.
- : Explicitly documented how non-role-prefixed line items are preserved.
- : Edited the README/docs to be clearer about how Releases works/operates.

- : Handful of typos, doc tweaks & addition of a .gitignore file. Thanks to Markus Zapke-Gründemann.
- : Fix a handful of bugs in release assignment logic.
- : Ensured Python 3 compatibility.
- : Fixed a stupid bug causing invalid issue hyperlinks.
- : Basic functionality.