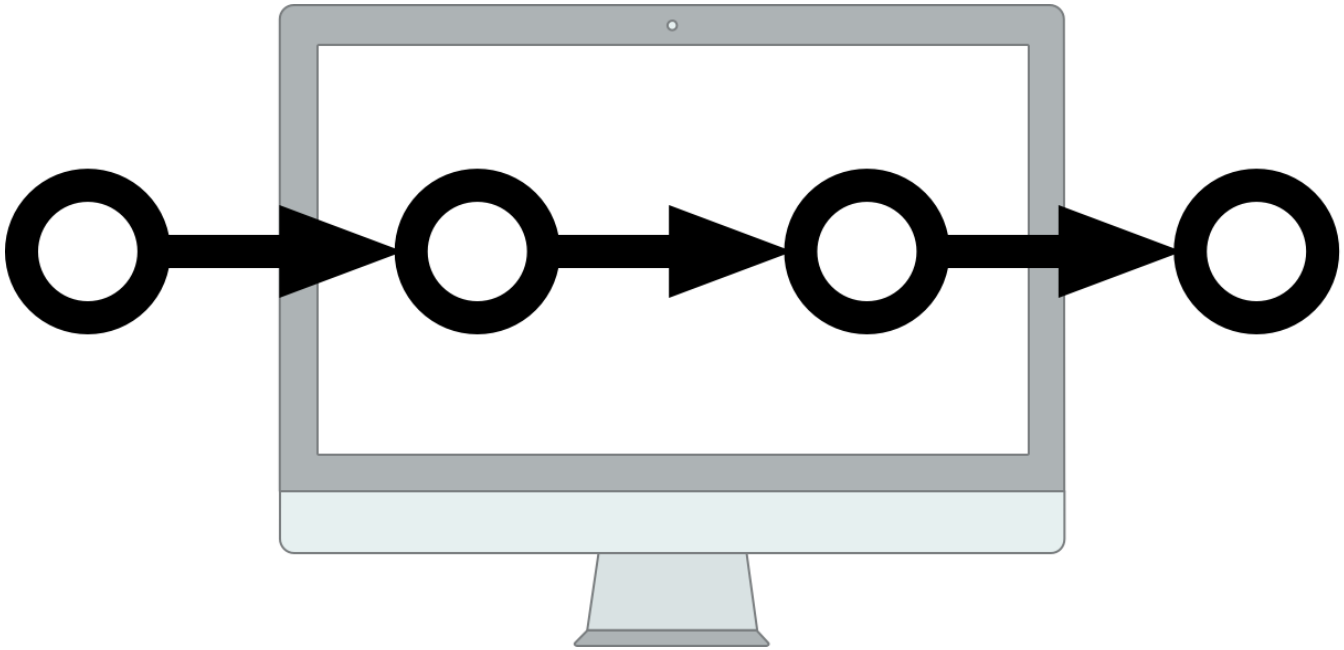
Semantic Versioning

Dominic Dumrauf

17.9.2017

How to Order Software Releases?



We need to version our software so that we can

- Establish an order among versions
- Eventually promote the correct version through the environments

Semantic Versioning

We can solve the problem of ordering software releases by using *semantic versioning*

Definition (Sub-versions)

A sub-version is a positive integer.

Definition (Semantic Versions)

A semantic version is

- A string of sub-versions, separated by dots
- Where the string starts with a sub-version and
- Where the string terminates with a sub-version

Semantic Versioning

We can solve the problem of ordering software releases by using *semantic versioning*

Definition (Sub-versions)

A sub-version is a positive integer.

Definition (Semantic Versions)

A semantic version is

- A string of sub-versions, separated by dots
- Where the string starts with a sub-version and
- Where the string terminates with a sub-version

Semantic Versions

Examples of Semantic Versions

- `0`
- `0.1`
- `1.1.25`
- `3.14.159.2653`

Examples of *non-Semantic Versions*

-
- `.`
- `.1`
- `1.`
- `1.1a.?5`

Semantic Versions

Examples of Semantic Versions

- `0`
- `0.1`
- `1.1.25`
- `3.14.159.2653`

Examples of *non-Semantic Versions*

-
- `.`
- `.1`
- `1.`
- `1.1a.?5`

Comparing Semantic Versions

Definition

Given a binary comparator operating on positive integers, two semantic versions are compared

- From left to right
- Sub-version by sub-version

Example

Given

- The binary comparator > and
- Two versions
  - `3.14.159`
  - `3.15.0`

Comparing Semantic Versions

Definition

Given a binary comparator operating on positive integers, two semantic versions are compared

- From left to right
- Sub-version by sub-version

Example

Given

- The binary comparator > and
- Two versions
  - `3.14.159`
  - `3.15.0`

Specifications

Comparing Semantic Versions

Implement `compare(version_1, version_2)` which

- Accepts two semantic versions `version_1` and `version_2`
- `version_1` contains a leading *comparator* out of a list of known comparators (=, >, and >=)
- `compare(version_1, version_2)` answers if "`version_2 version_1`"
- Return values are only `True` or `False`

Examples

- `compare('=0', '0') = True`
- `compare('>0', '1') = True`
- `compare('>=1.7.5, '1.7') = False`

Specifications

Comparing Semantic Versions

Implement `compare(version_1, version_2)` which

- Accepts two semantic versions `version_1` and `version_2`
- `version_1` contains a leading *comparator* out of a list of known comparators (=, >, and >=)
- `compare(version_1, version_2)` answers if "`version_2 version_1`"
- Return values are only `True` or `False`

Examples

- `compare('=0', '0') = True`
- `compare('>0', '1') = True`
- `compare('>=1.7.5, '1.7') = False`

Observations

Observation 1

Semantic versions can be regarded as

- Lists of positive integers
- Separated by dots

Observation 2

The special input `version_1` can be split into

- Comparator and
- Semantic version

by identifying the first number in the input.

Observations

Observation 1

Semantic versions can be regarded as

- Lists of positive integers
- Separated by dots

Observation 2

The special input `version_1` can be split into

- Comparator and
- Semantic version

by identifying the first number in the input.

Architectural Component Overview - Core Components

Version

- Parses a given version string and
- Provides binary comparator methods for semantic versions using magic methods

Comparator

- Knows the comparison symbol
- Compares two objects using native Python comparison
- For maximum reusability, bundle common functionality in a BasicComparator

Architectural Component Overview - Core Components

Version

- Parses a given version string and
- Provides binary comparator methods for semantic versions using magic methods

Comparator

- Knows the comparison symbol
- Compares two objects using native Python comparison
- For maximum reusability, bundle common functionality in a BasicComparator

Architectural Component Overview - The Glue

ComparatorAndVersionSeparator

Given a string, splits it into

- Comparator and
- Semantic version

VersionComparator

- Main entry point and
- Core orchestrator

Architectural Component Overview - The Glue

ComparatorAndVersionSeparator

Given a string, splits it into

- Comparator and
- Semantic version

VersionComparator

- Main entry point and
- Core orchestrator

Flow Inside VersionComparator

After passing it strings `version_1` and `version_2`, the component

1. Splits `version_1` string into comparator and semantic version `version_1` using a `ComparatorAndVersionSeparator`
2. Instantiate comparison object corresponding to comparator class
3. Parse `version_2` into semantic version `version_2`
4. Use comparator instance to compare both versions and return result

Errors

Should any of the above steps fail, then an exception is raised (yes, there are many exceptions for many different errors that can occurr)

Flow Inside VersionComparator

After passing it strings `version_1` and `version_2`, the component

1. Splits `version_1` string into comparator and semantic version `version_1` using a `ComparatorAndVersionSeparator`
2. Instantiate comparison object corresponding to comparator class
3. Parse `version_2` into semantic version `version_2`
4. Use comparator instance to compare both versions and return result

Errors

Should any of the above steps fail, then an exception is raised (yes, there are many exceptions for many different errors that can occurr)

Testing

Strategy

- Only test end-to-end, i.e. test `compare(version_1, version_2)`
- Ignore all implementation details on purpose
- Create a basic test framework covering all test cases that comparators can just hook into

Positive Tests

For each comparator, test as inputs all combinations of

1. One version being longer than the other one
2. Smaller, equal, or larger version numbers

Testing

Strategy

- Only test end-to-end, i.e. test `compare(version_1, version_2)`
- Ignore all implementation details on purpose
- Create a basic test framework covering all test cases that comparators can just hook into

Positive Tests

For each comparator, test as inputs all combinations of

1. One version being longer than the other one
2. Smaller, equal, or larger version numbers

Testing

Negative Tests

For each comparator, test as inputs

1. Improper input in either of the two versions
2. Missing comparator in `version_1`
3. Unknown comparator symbol

Bonus Tests

For each comparator test

1. That the comparator symbol matches the expected one
2. That a new comparator raises an error if misconfigured

Testing

Negative Tests

For each comparator, test as inputs

1. Improper input in either of the two versions
2. Missing comparator in `version_1`
3. Unknown comparator symbol

Bonus Tests

For each comparator test

1. That the comparator symbol matches the expected one
2. That a new comparator raises an error if misconfigured

Extendability By Design

Adding New Comparator

1. Start by using an existing comparator as a template
2. Add the comparator specific `SYMBOL` and `compare` method
3. Add comparator to `KNOWN_SYMBOLS_TO_COMPARATORS_MAPPING` in the `VersionComparator` class to make it available
4. Add corresponding magic method in `Version` class

Testing New Comparator

1. Start by using an existing comparator test as a template
2. Provide the expected comparator symbol
3. Provide the corresponding comparator class
4. Provide the expected results inside the helper methods

Extendability By Design

Adding New Comparator

1. Start by using an existing comparator as a template
2. Add the comparator specific `SYMBOL` and `compare` method
3. Add comparator to `KNOWN_SYMBOLS_TO_COMPARATORS_MAPPING` in the `VersionComparator` class to make it available
4. Add corresponding magic method in `Version` class

Testing New Comparator

1. Start by using an existing comparator test as a template
2. Provide the expected comparator symbol
3. Provide the corresponding comparator class
4. Provide the expected results inside the helper methods

Thank You!

Thanks For Your Attention!

Got Questions. . . ?