# Super State Machine Documentation

***Release 2.0.2***

**Szczepan Cieślik**

March 13, 2017

Contents


**1 Super State Machine** **3**
1.1 Features . . . 3

**2 Installation** **7**

**3 Usage** **9**
3.1 State machine . . . 9
3.2 Options . . . 12
3.3 State machine as property . . . 14
3.4 `utils` . . . 15

**4 API** **17**
4.1 *machines* . . . 17
4.2 *utils* . . . 17
4.3 *extras* . . . 17
4.4 *errors* . . . 17

**5 Contributing** **19**
5.1 Types of Contributions . . . 19
5.2 Get Started! . . . 20
5.3 Pull Request Guidelines . . . 20
5.4 Tips . . . 21

**6 Credits** **23**
6.1 Development Lead . . . 23
6.2 Contributors . . . 23

**7 History** **25**
7.1 2.0.2 (2017-03-13) . . . 25
7.2 2.0.1 (2017-02-27) . . . 25
7.3 2.0 (2016-09-26) . . . 25

**8 1.0 (2014-09-04)** **27**

**9 0.1.0 (2014-08-08)** **29**

**10 Indices and tables** **31**

**Python Module Index** **33**

Contents:

CHAPTER 1

# Super State Machine

Super State Machine gives you utilities to build finite state machines.

- Free software: BSD license
- Documentation: https://super_state_machine.readthedocs.org
- Source: https://github.com/beregond/super_state_machine

## Features

- Fully tested with Python 2.7, 3.3, 3.4 and PyPy.
- Create finite state machines:

```
>>> from enum import Enum

>>> from super_state_machine import machines


>>> class Task(machines.StateMachine):
...
...     state = 'draft'
...
...     class States(Enum):
...
...         DRAFT = 'draft'
...         SCHEDULED = 'scheduled'
...         PROCESSING = 'processing'
...         SENT = 'sent'
...         FAILED = 'failed'

>>> task = Task()
>>> task.is_draft
False
>>> task.set_draft()
>>> task.state
'draft'
>>> task.state = 'scheduled'
>>> task.is_scheduled
True
>>> task.state = 'process'
>>> task.state
```

```
'processing'
>>> task.state = 'wrong'
*** ValueError: Unrecognized value ('wrong').
```

- Define allowed transitions graph, define additional named transitions and checkers:

```
>>> class Task(machines.StateMachine):
...
...     class States(Enum):
...
...         DRAFT = 'draft'
...         SCHEDULED = 'scheduled'
...         PROCESSING = 'processing'
...         SENT = 'sent'
...         FAILED = 'failed'
...
...     class Meta:
...
...         allow_empty = False
...         initial_state = 'draft'
...         transitions = {
...             'draft': ['scheduled', 'failed'],
...             'scheduled': ['failed'],
...             'processing': ['sent', 'failed']
...         }
...         named_transitions = [
...             ('process', 'processing', ['scheduled']),
...             ('fail', 'failed')
...         ]
...         named_checkers = [
...             ('can_be_processed', 'processing'),
...         ]

>>> task = Task()
>>> task.state
'draft'
>>> task.process()
*** TransitionError: Cannot transit from 'draft' to 'processing'.
>>> task.set_scheduled()
>>> task.can_be_processed
True
>>> task.process()
>>> task.state
'processing'
>>> task.fail()
>>> task.state
'failed'
```

Note, that third argument restricts from which states transition will be **added** to allowed (in case of `process`, new allowed transition will be added, from 'scheduled' to 'processing'). No argument means all available states, `None` or empty list won't add anything beyond defined ones.

- Use state machines as properties:

```
>>> from enum import Enum

>>> from super_state_machine import machines, extras
```

```
>>> class Lock(machine.StateMachine):

...     class States(Enum):
...
...         OPEN = 'open'
...         LOCKED = 'locked'
...
...     class Meta:
...
...         allow_empty = False
...         initial_state = 'locked'
...         named_transitions = [
...             ('open', 'open'),
...             ('lock', 'locked'),
...         ]


>>> class Safe(object):
...
...     lock1 = extras.PropertyMachine(Lock)
...     lock2 = extras.PropertyMachine(Lock)
...     lock3 = extras.PropertyMachine(Lock)
...
...     locks = ['lock1', 'lock2', 'lock3']
...
...     def is_locked(self):
...          locks = [getattr(self, lock).is_locked for lock in self.locks]
...          return any(locks)
...
...     def is_open(self):
...         locks = [getattr(self, lock).is_open for lock in self.locks]
...         return all(locks)

>>> safe = Safe()
>>> safe.lock1
'locked'
>>> safe.is_open
False
>>> safe.lock1.open()
>>> safe.lock1.is_open
True
>>> safe.lock1
'open'
>>> safe.is_open
False
>>> safe.lock2.open()
>>> safe.lock3 = 'open'
>>> safe.is_open
True
```

CHAPTER 2

# Installation

At the command line:

```
$ easy_install super_state_machine
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv super_state_machine
$ pip install super_state_machine
```

CHAPTER 3

# Usage

## State machine

State machine allows to operate on state, where allowed states are defined by states enum.

Remember that states enum **must be** unique.

### Meta

All options for state machine are passed through `Meta` class, like below:

```
>>> class Task(machines.StateMachine):
...
...     class States(Enum):
...
...         DRAFT = 'draft'
...         SCHEDULED = 'scheduled'
...         PROCESSING = 'processing'
...         SENT = 'sent'
...         FAILED = 'failed'
...
...     class Meta:
...
...         named_checkers = [
...             ('can_be_processed', 'processing'),
...         ]
```

You can see that about only option `named_checkers` is provided. In fact it is not necessary to provide any option at all. For full reference see *Options*.

### Word about value translation

Whenever you will be passing enum value or string to represent state (in meta, in options, in methods `is_*`, `set_*` or `can_be_*`) remember that these values must **clearly** describe enum value.

For example in following case:

```
>>> class Lock(machine.StateMachine):
...
...     class States(Enum):
...
```

```
...         OPEN = 'open'
...         OPENING = 'opening'
...         LOCKED = 'locked'
...         LOCKING = 'locking'
```

values that clear state `open` are string `'open'` and `Lock.States.OPEN`, but for `opening` state these are strings `'openi'`, `'opening'` and `Lock.States.OPENING`. In other words you must provide as much information to make it not necessary to guess end value. Otherwise `AmbiguityError` will be raised.

## Simple case

In simplest case you just have to define `States` enum to definen what valid states are and start using it.

```
>>> from enum import Enum

>>> from super_state_machine import machines


>>> class Task(machines.StateMachine):
...
...     class States(Enum):
...
...         DRAFT = 'draft'
...         SCHEDULED = 'scheduled'
...         PROCESSING = 'processing'
...         SENT = 'sent'
...         FAILED = 'failed'

>>> task = Task()
>>> task.is_draft
False
>>> task.set_draft()
>>> task.state
'draft'
>>> task.state = 'scheduled'
>>> task.is_scheduled
True
>>> task.state = 'p'
>>> task.state
'processing'
>>> task.state = 'wrong'
*** ValueError: Unrecognized value ('wrong').
```

## Actual state as enum

You can also get actual state in enum form by property `actual_state`, or `as_enum`:

```
>>> task.actual_state
<States.DRAFT: 'draft'>
>>> task.as_enum
<States.DRAFT: 'draft'>
```

## Transitions

In case when you want to define what proper transitions are, you need to define `transitions` option.

```
>>> class Task(machines.StateMachine):
...
...     class States(Enum):
...
...         DRAFT = 'draft'
...         SCHEDULED = 'scheduled'
...         PROCESSING = 'processing'
...         SENT = 'sent'
...         FAILED = 'failed'
...
...     class Meta:
...
...         transitions = {
...             'draft': ['scheduled', 'failed'],
...             'scheduled': ['failed'],
...             'processing': ['sent', 'failed'],
...         }
...         named_transitions = [
...             ('process', 'processing', ['scheduled']),
...             ('fail', 'failed'),
...         ]
```

In example above `transitions` option defines which transitions are valid - for example from that option we can read that state can be switched to `draft` but only from `scheduled` or `failed`.

You can change state to desired one by generated methods like `set_*`, so if you want to change state of `Task` to `draft` it is enough to call `set_draft` on instance of `Task`.

There is also `named_transitions` option. This is list of 3-tuples with name, desired state optional "from" states, or 2-tuples with name and desired states. First line means that instance of task will have method called `process` which will trigger change of state to `process`. It is like you would call method `set_processing` but sounds better. Also all "from" states are **added** to list of valid transitions of `Task`.

> **Warning:** In case you won't provide third argument in tuple, it is considered that transition to that case is allowed from ANY other state (like `('fail', 'failed')` case). If you want just to add named transition without modifying actual transitions table, pass as `None` as third argument.
>
> ```
> ...     named_transitions = [
> ...         ('process', 'processing', None),
> ...     }
> ```

**See also:**

*complete*

## Forced set (forced transition)

You can also use `force_set` which will change current state to any other **proper** state without checkint if such transition is allowed. It may be seen as 'hard reset' to some state.

```
>>> task.force_set('draft')
>>> task.force_set(Task.States.SCHEDULED)
```

New in version 2.0.

### Checkers

```
>>> class Task(machines.StateMachine):
...
...     class States(Enum):
...
...         DRAFT = 'draft'
...         SCHEDULED = 'scheduled'
...         PROCESSING = 'processing'
...         SENT = 'sent'
...         FAILED = 'failed'
...
...     class Meta:
...
...         named_checkers = [
...             ('can_be_processed', 'processing'),
...         ]
```

Each instance of state machine has auto generated set of checkers (which are properties) like `can_be_*`. In this case checkers will be like `can_be_draft`, `can_be_sent` etc. If you want to have custom checkers defined, you can either define them by yourself or pass as 2-tuple in `named_checkers` option. Tuple must have name of checker and state to check, so in this case instance of `Task` will have property `can_be_processed` which will work like `can_be_processing` (yet sounds better).

### Getters

```
>>> class Task(machines.StateMachine):
...
...     class States(Enum):
...
...         DRAFT = 'draft'
...         SCHEDULED = 'scheduled'
...         PROCESSING = 'processing'
...         SENT = 'sent'
...         FAILED = 'failed'
```

Getters checks state, but checks one particular state. All of getters are properties and are named like `is_*`. If you want to check if instance of `Task` is currently draft, just call `instance.is_draft`. This work just like calling `instance.is_('draft')`. This comes handy especially in templates.

### Name collisions

In case any auto generated method would collide with already defined one, or if named transitions or checkers would cause collision with already defined one or with other auto generated method, `ValueError` will be raised. In particular name collisions (intentional or not) are prohibited and will raise an exception.

## Options

#### `states_enum_name`

Default value: `'States'`.

Define name of states enum. States enum must be present in class definition under such name.

### allow_empty

Default value: `True`.

Determine if empty state is allowed. If this option is set to `False` option *initial_state* **must** be provided.

### initial_state

Default value: `None`.

Defines initial state the instance will start it's life cycle.

### complete

This option defines if states **graph** is complete. It this option is set to `True` then any transition is **always** valid. If this option is set to `False` then state machine looks to states graph to determine if this transition should succeeed.

This option in fact doesn't have default value. If isn't provided and `transitions` neither `named_transitions` options are not provided then it is set to `True`. If one or both options are provided this option is set to `False` (still, only if it wasn't provided in `Meta` of state machine).

### transitions

Dict that defines basic state graph (which can be later filled up with data comming from *named_transitions*).

Each key defines target of transition, and value (which **must** be a list) defines initial states for transition.

```
...     class Meta:
...
...         transitions = {
...             'draft': ['scheduled', 'failed'],
...             'scheduled': ['failed'],
...             'processing': ['sent', 'failed'],
...         }
```

### named_transitions

List of 3-tuples or 2-tuples (or mixed) which defines named transitions. These definitions affect states graph:

- If there is no third argument (2-tuple was passed) then desired transition is valid from **all** states.
- If there is `None` passed as third argument - the states **will not** be affected.
- Otherwise third argument must be list of allowed initial states for this transition. Remember that these transitions will be **added** to state graph. Also other transitions defined in *transitions* option will still be valid for given transition name.

```
...     class Meta:
...
...         transitions = {
...             'draft': ['scheduled', 'failed'],
...             'scheduled': ['failed'],
...             'processing': ['sent', 'failed'],
...         }
...         named_transitions = [
```

```
...             ('process', 'processing', ['scheduled']),
...             ('fail', 'failed'),
...         ]
```

In this case method `process` will change state to `processing` but transition is valid from three initial states: `scheduled`, `sent` and `failed`.

### named_checkers

List of 2-tuple which defines named transition checkers. Tuple consist of checker name and desired state. When called, checher will check if state machine can transit to desired state.

```
...     class Meta:
...
...         named_checkers = [
...             ('can_be_processed', 'processing'),
...         ]
```

In example above property `can_be_processed` on instance will determine if state can be changed to state `processing`.

## State machine as property

Thanks to `extras` module you can use state machines as properties!

```
>>> from enum import Enum

>>> from super_state_machine import machines, extras


>>> class Lock(machine.StateMachine):

...     class States(Enum):
...
...         OPEN = 'open'
...         LOCKED = 'locked'
...
...     class Meta:
...
...         allow_empty = False
...         initial_state = 'locked'
...         named_transitions = [
...             ('open', 'o'),
...             ('lock', 'l'),
...         ]


>>> class Safe(object):
...
...     lock1 = extras.PropertyMachine(Lock)
...     lock2 = extras.PropertyMachine(Lock)
...     lock3 = extras.PropertyMachine(Lock)
...
...     _locks = ['lock1', 'lock2', 'lock3']
...
```

```
...     def is_locked(self):
...          locks = [getattr(self, lock).is_locked for lock in self._locks]
...          return any(locks)
...
...     def is_open(self):
...         locks = [getattr(self, lock).is_open for lock in self._locks]
...         return all(locks)

>>> safe = Safe()
>>> safe.lock1
'locked'
>>> safe.is_open
False
>>> safe.lock1.open()
>>> safe.lock1.is_open
True
>>> safe.lock1
'open'
>>> safe.is_open
False
>>> safe.lock2.open()
>>> safe.lock3 = 'open'
>>> safe.is_open
True
```

In this case method `as_enum` is really handy:

```
>>> safe.lock1.as_enum
<States.OPEN: 'open'>
```

Although you could also use `actual_state` here (yet `as_enum` sounds more familiar).

> **Warning:** In this case value is always visible as string, so there is **no** `None` value returned. Instead of this `None` is transformed into `''` (empty string).

---

**Note:** Remember that change of state can be made by calling method `safe.lock1.lock`, assignation of string (or its part) like `safe.lock1 = 'open'` or `safe.lock1 = 'o'` or assignation of enum like `safe.lock1 = Lock.States.OPEN`.

---

## utils

### EnumValueTranslator

This class is part of inner API (see `super_state_machine.utils.Enumvaluetranslator`) but is really handy - it is used by state machine to translate all (short) string representations to enum values.

It also can ensure that given enum belongs to proper states enum.

```
>>> import enum

>>> from super_state_machine import utils


>>> class Choices(enum.Enum):
```

```
...
...     ONE = 'one'
...     TWO = 'two'
...     THREE = 'three'


>>> class OtherChoices(enum.Enum):
...
...    ONE = 'one'

>>> trans = utils.Enumvaluetranslator(Choices)
>>> trans.translate('o')
<Choices.ONE: 'one'>
>>> trans.translate('one')
<Choices.ONE: 'one'>
>>> trans.translate(Choices.ONE)
<Choices.ONE: 'one'>

>>> trans.translate('t')
*** AmbiguityError: Can't decide which value is proper for value 't' (...)

>>> trans.translate(OtherChoices.ONE)
*** ValueError: Given value ('OtherChoices.ONE') doesn't belong (...)
```

CHAPTER 4

# API

Contents:

## machines

## utils

## extras

Extra utilities for state machines, to make them more usable.

**class** `super_state_machine.extras.`**`PropertyMachine`**(*machine_type*)
Descriptor to help using machines as properties.

**class** `super_state_machine.extras.`**`ProxyString`**
String that proxies every call to nested machine.

## errors

Errors module.

**exception** `super_state_machine.errors.`**`TransitionError`**
Raised for situation, when transition is not allowed.

CHAPTER 5

# Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

## Types of Contributions

### Report Bugs

Report bugs at https://github.com/beregond/super_state_machine/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

### Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

### Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

### Write Documentation

Super State Machine could always use more documentation, whether as part of the official Super State Machine docs, in docstrings, or even on the web in blog posts, articles, and such.

### Submit Feedback

The best way to send feedback is to file an issue at https://github.com/beregond/super_state_machine/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

## Get Started!

Ready to contribute? Here's how to set up *super_state_machine* for local development.

1. Fork the *super_state_machine* repo on GitHub.
2. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/super_state_machine.git
   ```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

   ```
   $ mkvirtualenv super_state_machine
   $ cd super_state_machine/
   $ python setup.py develop
   ```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

   ```
   $ flake8 super_state_machine tests
   $ python setup.py test
   $ tox
   ```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.

## Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, and 3.4, and for PyPy. Check https://travis-ci.org/beregond/super_state_machine/pull_requests and make sure that the tests pass for all supported Python versions.

## Tips

To run a subset of tests:

```
$ python -m unittest tests.test_super_state_machine
```

CHAPTER 6

# Credits

## Development Lead

- Szczepan Cieślik <szczepan.cieslik@gmail.com>

## Contributors

(In alphabetical order)

- Eric Dill <eric.dill@maxpoint.com>
- Thomas A Caswell <tcaswell@gmail.com>

CHAPTER 7

# History

## 2.0.2 (2017-03-13)

- Fixed requirements for Python > 3.4.

## 2.0.1 (2017-02-27)

- Remove enum34 for Python > 3.4.
- Added support for Python 2.6.

## 2.0 (2016-09-26)

- Added force_set method.
- Added field machine.
- Added support for Python 3.5.

Backward compatibility breaks:

- Empty state is now disallowed.
- Only full names are allowed, when using scalars, no shortcuts.
- Removed support for unhashable types.

CHAPTER 8

# 1.0 (2014-09-04)

- Added all basic features.

CHAPTER 9

# 0.1.0 (2014-08-08)

- First release on PyPI.
- Added utilities to create simple state machine.

CHAPTER 10

# Indices and tables

- *genindex*
- *modindex*
- *search*

# Python Module Index

## S

super_state_machine.errors, 17
super_state_machine.extras, 17

## P

PropertyMachine (class in super_state_machine.extras), 17
ProxyString (class in super_state_machine.extras), 17

## S

super_state_machine.errors (module), 17
super_state_machine.extras (module), 17

## T

TransitionError, 17