PyKleen Documentation

Release 1.4

Nils Corver

Dec 16, 2016

Contents

Input; simple, clean.

CHAPTER 1

Example

This is how you do it.

```
# -*- coding: utf-8 -*-
import pykleen as k

schema = k.Schema({
    'name': k.string(min_len=1),
    'age': k.numeric(),
    'addresses': k.list_of(k.Schema({
        'number': k.numeric(),
        'postal': k.string(min_len=4, max_len=8),
    })),
})

person = schema({
    'name': 'Kleener',
    'age': '38',
    'addresses': [{
        'number': '40',
        'postal': '67768GG',
    }]
})
```

There are more tests and you can build your own! See the API Reference for more information.

CHAPTER 2

API Reference

2.1 Tests

2.1.1 Schema

class Schema

Schema is a *Param* -> *Test* container. This is usually your starting point, as most information you need to _kleen_ is user input in the form of a dictionary.

class Param

When defining a param in the *Schema*, you can use the *Param* class to mark the parameter as required or attach a default.

2.1.2 Tests

string()

Sanitize and validate a value as a string. When using a users input, HTML tags in the input are always a critical point. This function uses, by default, the *bleach* package to strip tags from the input.

```
test = string(10, 150, strip_tags=True, allowed_tags=['a', 'p'])
value = test('Some test here')  # 'Some test here'
value = test('Some')  # 'Some test here'
value = test(23)  # <Error 'Should not exceed 10.'>
```

Parameters

- **min_len** (*int = None*) – Minimum length of the string.
- **max_len** (*int = None*) – Maximum length of the string.
- **strip_tags** (*bool = True*) – Should HTML-tags be stripped?
- **allowed_tags** (*List[str] = None*) – Which tags are allowed?
- **allowed_attributes** (*Dict[str, List[str]] = None*) – Which attributes are allowed?
- **strip** (*bool = True*) – Should whitespace be stripped?
- **min_len_error** (*str = 'Too short.'*) – Error for *min_len*.
- **max_len_error** (*str = 'Too long.'*) – Error for *max_len*.

regex()

Validate a string based on the given regular expression. This test uses *string()* by default, but you can substitute your own.

```
test = regex(r'^[a-z]+$', error='Only letters.')
value = test('testtest')  # 'testtest'
value = test(23)  # <Error 'Only letters.'>
```

Parameters

- **pattern** (*str*) – Regular expression.
- **cast_to_string** (*Callable = None*) – Cast to string, ran before the pattern test.
- **error** (*str = 'Does not match.'*) – Error

email()

Validates an email address based on the RFC 2822 specifications. Also the email address should be at least 5 characters and at most 256 characters.

```
test = email()
value = test('example@domain.com')  # 'example@domain.com'
value = test('test.com')  # <Error 'Invalid email address.'>
```

Parameters **error** (*str = 'Does not match.'*) – Error

password()

Validates a password on a set of rules. The score of the value should be at least the required strength (default 70). It uses 14 tests to determine the strength.

```
test = password()
value = test('Test123!')  # 'Test123!'
value = test('test')  # <Error 'Password is too weak.'>
```

Parameters

- **strength** (*int = 70*) – Minimum required strength of the password.
- **too_weak_error** (*str = 'Password is too weak.'*) – Error

numeric()

Validate and sanitize a numeric value. The result will always be a *Decimal* and will be rounded *half up* by default.

```
test = numeric()
value = test('23')  # Decimal('23')
value = test('24,7')  # Decimal('23')

test = numeric(3)
value = test('23')  # Decimal('23.000')
value = test('24.012,209')  # Decimal('24012.209')
```

Parameters

- **decimals** (*int = 0*) – Amount of decimals allowed.
- **rounding** (*str = ROUND_HALF_UP,*) – How rounding should be applied, use *decimal.ROUND_**.

- **at_least** (*int = None*) – Lowest allowed value.
- **at_most** (*int = None,*) – Highest allowed value.
- **at_least_error** (*str = 'Too small.',*) – Error for *at_least*.
- **at_most_error** (*str = 'Too large.',*) – Error for *at_most*.
- **cast_error** (*str = 'Invalid number.'*) – Error when casting do Decimal failed.

boolean()

Turn any value to a boolean (via *str*). You can provide your own lists to determine True-ness or False-ness. By default this function casts to true when *['True', 'true', '1', 'yes', 'y', 'on']* otherwise the result is *False*.

```
test = boolean()
value = test('yes')  # True
value = test('nope')  # False
value = test(True)  # True

test = boolean(is_false_when=['no', 'False', '0'])
value = test('no')  # False
value = test('yepper')  # True
value = test(None)  # True
```

Parameters

- **is_true_when** (*List[str] = None*) – List of *True* values.
- **is_false_when** (*List[str] = None*) – List of *False* values.
- **cast_error** (*str = 'Invalid boolean.'*) – When *is_true_when* and *is_false_when* are not matched. Only applicable when you provide both lists!

datetime()

Turn a value into a *datetime.datetime* value. You can provide your own list of formats to parse against. By default is is this list of formats:

- '%Y-%m-%d %H:%M'
- '%d-%m-%Y %H:%M'
- '%Y-%m-%d %H:%M:%S'
- '%d-%m-%Y %H:%M:%S'
- '%Y-%m-%dT%H:%M:%S'

```
test = datetime()
value = test('2001-4-23 23:01')  # datetime.datetime(2001, 4, 23, 23, 1)
value = test('not a date')  # <Error 'Invalid datetime.'>
```

Parameters

- **formats** (*List[str] = None*) – A list of formats used with *strptime*.
- **parse_error** (*str = 'Invalid datetime.'*) – Error when no format is matched.

date()

Turn a value into a *datetime.date* value. You can provide your own list of formats to parse against. By default is is this list of formats:

- `'%Y-%m-%d'`
- `'%d-%m-%Y'`

```
test = date()
value = test('2001-4-23')  # datetime.date(2001, 4, 23)
value = test('not a date')  # <Error 'Invalid date.'>
```

Parameters

- **formats** (*List[str] = None*) – A list of formats used with *strptime*.
- **parse_error** (*str = 'Invalid date.'*) – Error when no format is matched.

minutes()

Turn a value into an *int* value. It will parse (at least it tries) some formats people use for minutes.

- `'hh:mm'`, `'hhhhh:mm'`
- `'hh:mm'`, `'hh.mm'`, `'hh,mm'`
- `'hhmm'`, `'hmm'`
- `'mmmmm'`

```
test = minutes()
value = test('10:33')  # 633
value = test('10.35')  # 635
value = test('1028')  # 628
value = test('not a date')  # <Error 'Invalid date.'>
```

Parameters **error** (*str = 'Invalid format.'*) – Error when no format is matched.

one_of()

Check if the value is one of the defined values. You can use the *cast* parameter to cast the value before checking, for when you want to check numbers or booleans.

```
test = one_of(['a', 'b', 'c', '1', '2'], cast=string())
value = test('b')  # 'b'
value = test(2)  # '2'
value = test('8')  # <Error 'Invalid value.'>
```

Parameters

- **values** (*List[Any]*) – List of values it should match to.
- **cast** (*Union[Callable, Test] = None*) – Values in the list can be cast before checking.
- **invalid_error** (*str = 'Invalid value.'*) –

set_of()

Check if all the item in the value, which is a list, set or tuple, are present in the defined values. You can use the *cast* parameter to cast each item before checking, for when you want to check numbers or booleans.

See *convert_to_list()* for more details.

```
test = set_of(['a', 'b', 'c', '1', '2'], cast=string())
value = test('b')  # ['b']
value = test('a,2')  # ['a', '2']
value = test(['a', 'c', 1])  # ['a', 'c', '1']
value = test('7,1,8')  # <Error 'Invalid value(s) 7, 8.'>
```

Parameters

- **values** (*List[Any]*) – List of values it should match to.
- **cast** (*Union[Callable, Test] = None*) – Values in the list can be cast before checking.
- **invalid_error** (*str = 'Invalid value.'*) –

list_of()

The passed value should be a list of *cast*. You can use this to get a list of numbers of a list of booleans, but you can also use this in combination with *Schema*.

See *convert_to_list()* for more details.

```
test = list_of(numeric())
value = test(['4', '18,2'])  # [Decimal('4'), Decimal('18')]
value = test('8,12')  # [Decimal('8'), Decimal('12')]
value = test(['a', '13', 'c'])  #  <Errors [0, 2] Invalid item(s).>
```

Parameters

- **cast** (*Union[Callable, Test]*) – Used to cast each item in the value.
- **error** (*str = 'Invalid item(s) %s.'*) – Error message when one or more casts fail.
- **type_error** (*str = 'Invalid type %s.'*) – Error when an invalid value type is provided.

2.1.3 Others

convert_to_list()

Convert a value into a list, this should be used as an helper function as the value it produces contains unchecked items.

•It splits a string on `','`, so you can pass values like `'1,3,5'`.

•Values that are `int`, `Decimal`, `.datetime` or `.date` are wrapped by a list.

•Any other list type is accepted, other values throw an error.

```
def list_of_numbers():
  return All([
    convert_to_list(),
    To(numeric()),
  ])

test = list_of_numbers()
value = test('1,2')  # [Decimal('1'), Decimal('2')]
```

is_instance()
Verify the value is of a certain instance. You can use this to verify if the value is correctly converted or passed. It does not (in any way) sanitize the value, it only checks.

```
def should_be_file():
  return is_instance(file)

test = should_be_file()

stream = open('f.txt')
value = test(stream)  # <_io.TextIOWrapper ...>
value.close()
```

Parameters

- **types** (*List[Any]*) – A list of types to validate against.
- **type_error** (*str = 'Invalid type %s.'*) – Error if the value is not of instance.

size()
Verify if the item is of a certain size, this function is used by the *string()* function. You could use it in conjunction with *set_of()* or *list_of()*, to check the size of the list.

```
def list_of_numbers(min_len=None, max_len=None):
  return All([
    list_of(numeric()),
    size(min_len, max_len),
  ])

test = list_of_numbers(3, 5)
value = test('1,2,3')  # [Decimal('1'), Decimal('2'), Decimal('3')]
value = test('1')  # <Error 'Too short.'>
value = test('1,3,5,6,7,8')  # <Error 'Too long.'>
```

Parameters

- **min_len** (*int = None*) – Minimum size of the value.
- **max_len** (*int = None*) – Maximum size of the value.
- **min_len_error** (*str = 'Too short.'*) – Error for *min_len*.
- **max_len_error** (*str = 'Too long.'*) – Error for *max_len*.
- **type_error** (*str = 'Invalid type %s.'*) – Error for not being a string.

2.2 Structure

You can build you own tests, validators and sanitizers using these classes.

class Test
A test can be called and will yield a validated and/or sanitized product. The *Test* itself doesn't do anything.

class One
Given a callback, when the *One* test is ran, it runs the callback with the given value.

```
test = One(lambda v: v < 10, 'Should not exceed 10.')
value = test(6)  # 6
value = test(23)  # <Error 'Should not exceed 10.'>
```

__init__(*callback*[, *error*])

Parameters

- **callback** (*Union[Callable, Test]*) – The test that runs when called.
- **error** (*str*) – Value of the error-message (*'Test failed.'*).

class All

Given multiple callbacks in a list, when the *All* test is ran, it runs the all the callbacks with the given value.

Be aware that any sanitization must be wrapped in a *To*, when providing a simple function the outcome is not saved. A lambda (or any other callback like `str` or `int`) is only tested on truthness, when false is throws the *error*.

```
test = All([To(int), lambda v: v < 10], 'Must be an int, less than 10.')
value = test('6')  # 6
value = test(23)  # <Error 'Must be an int, less than 10.'>
value = test('aa')  # <Error 'Must be an int, less than 10.'>
```

__init__(*callbacks*[, *error*])

Parameters

- **callbacks** (*List[Union[Callable, Test]]*) – The test that runs when called.
- **error** (*str*) – Value of the error-message (*'Test failed.'*).

class Or

Given multiple callbacks in a list, when the *Or* test is ran, it runs the callbacks one by one until a success (errors are discarded). The successful result is returned. When no callback is successful an error is thrown.

```
test = Or([date(), datetime(), numeric()], 'Requires date, datetime or␣
↪timestamp (int).')
value = test('2004-2-18')  # datetime.date(2004, 2, 18)
value = test(1077062400)  # Decimal(1077062400)
value = test('aa')  # <Error 'Requires date, datetime or timestamp (int).'>
```

__init__(*callbacks*[, *error*])

Parameters

- **callbacks** (*List[Union[Callable, Test]]*) – A list of callbacks.
- **error** (*str = 'All tests failed.'*) – Error when none of the callbacks is successful.

class To

This is a sanization helper, it simply tries to convert a value *To* another value. Usefull when running *All*, mostly in conjunction with other tests.

```
test = To(int)
value = test('6')  # 6
value = test(2.4)  # 2
value = test('a')  # <Error 'Conversion failed.'>
```

__init__(*callbacks*[, *error*])

Parameters

- **into** (*Union[Callable, Test]*) – Turn the value *into* this.
- **error** (*str = 'Conversion failed.'*) – Error when the conversion failed.

class Yield

Yield a fixed result when the callback over the value succeeds. There are not many usecases for this, but in conjunction with *Or* it can be handy (providing default values on test-success).

```
test = Or([
  Yield(lambda v: v in [True, False], 'boolean'),
  Yield(lambda v: v in [None], 'none'),
  Yield(One(lambda v: isinstance(v, (str, bytes))), 'string'),
  Yield(One(lambda v: isinstance(v, int)), 'number'),
])

value = test(True) # 'boolean'
test(None) # 'none'
test('blurp') # 'string'
test(44) # 'number'
```

__init__(*callbacks*[, *error*])

Parameters

- **callback** (*Union[Callable, Test]*) – The test to run.
- **fixed_result** (*Any*) – Returned when the callback is truthful.
- **error** (*str = 'Test failed.'*) – Error when the callback was not truthful.

2.3 Development

Some simple instructions for the development of PyKleen.

2.3.1 Setup

Setup your virtual environment, activate it and install the requirements.

```
virtualenv --no-site-packages -p `which python3.5` venv
source venv/bin/activate
pip install -r requirements.txt
deactivate && source venv/bin/activate
```

2.3.2 Development

While doing your development run:

```
PYTHONPATH=. ptw
```

Or, when you don't want to keep the test running:

```
python setup.py test
```

After you are finished, you need to update the documentation:

```
cd docs && make livehtml
```

2.3.3 Release

1. Update the version number in *VERSION*.
2. Run the tests once more (better to be sure, right?).
3. Commit and push to GIT.
4. Release the code.

```
echo "X.X" > VERSION
python setup.py test
ga . && gc -m 'Bump to version X.X' && gp
python setup.py sdist upload -r pypi
```

Symbols

A

B

C

D

E

I

L

M

N

O

P

R

S

T

Y