Verify Documentation

Release 1.1.1

Derrick Gilland

May 09, 2017

Contents

Verify is a painless assertion library for Python.

CHAPTER 1

Links

- Project: https://github.com/dgilland/verify
- Documentation: http://verify.readthedocs.org
- PyPI: https://pypi.python.org/pypi/verify/
- TravisCI: https://travis-ci.org/dgilland/verify

CHAPTER 2

Quickstart

Install using pip:

```
pip install verify
```

Verify some value using multiple assertions:

```
from verify import expect, Not, Truthy, Falsy, Less, Greater

expect(5 * 5,
       Truthy(),
       Not(Falsy),
       Greater(15),
       Less(30))
```

Verify using your own assert functions:

```
def is_just_right(value):
    assert value == 'just right', 'Not just right!'

# Passes
expect('just right', is_just_right)

# Fails
try:
    expect('too cold', is_just_right)
except AssertionError:
    raise
```

NOTE: The assert function should return a truthy value, otherwise, `expect` will treat the falsy return from the function as an indication that it failed and subsequently raise it's own `AssertionError`.

Verify using your own predicate functions:

```
def is_awesome(value):
    return 'awesome' in value

def is_more_awesome(value):
    return value > 'awesome'

expect('so awesome', is_awesome, is_more_awesome)
```

Verify using chaining syntax:

```
expect(1).Truthy().Number().NotBoolean().Not(is_awesome)
```

Verify without `expect` since the `verify` assertions can be used on their own:

```
import verify

# These would pass.
verify.Truthy(1)
verify.Equal(2, 2)
verify.Greater(3, 2)

# These would fail with an AssertionError
verify.Truthy(0)
verify.Equal(2, 3)
verify.Greater(2, 3)
```

If you'd prefer to see `assert` being used, all `verify` assertions will return `True` if no `AssertionError` is raised:

```
assert Truthy(1)
assert expect(1, Truthy(), Number())
```

CHAPTER 3

Multiple Syntax Styles

There are several syntax styles available to help construct more natural sounding assertion chains.

Expect...To Be

Use `expect` with the `to_be` aliases. All Pascal case assertions have `to_be_*` and `to_not_be_*` prefixes (with a few expections).

```
expect(something).to_be_int().to_be_less_or_equal(5).to_be_greater_or_equal(1)
expect(something_else).to_not_be_float().to_be_number()
```

Ensure...Is

Use `ensure` with `is` aliases. All Pascal case assertions have `is_*` and `is_not_*` prefixes (with a few expections).

```
ensure(something).is_int().is_less_or_equal(5).is_greater_or_equal(1)
ensure(something_else).is_not_float().is_number()
```

Classical

Use `expect` or `ensure` with the Pascal case assertions.

```
ensure(something).Int().LessOrEqual(5).GreaterOrEqual(1)
expect(something_else).Float().Number()
```

NOTE: While it's suggested to not mix styles, each of the assertion syntaxes are available with both `expect` and `ensure`. So you can call `expect(..).is_int()` as well as `ensure(..).to_be_int()`.

Naming Convention Exceptions

As mentioned above, there are some assertions that have nonstandard aliases:

- `Not`: `not_`, `does_not`, `to_fail`, and `fails`
- `Predicate`: `does`, `to_pass`, and `passes`

- `All`: `all_`, `does_all`, and `passes_all`
- `NotAll`: `not_all`, `does_not_all`, and `fails_all`
- `Any`: `any_`, `does_any`, and `passes_any`
- `NotAny`: `not_any`, `does_not_any`, and `fails_any`
- `Match`: `to_match`, `is_match` and `matches`
- `NotMatch`: `to_not_match`, `is_not_match` and `does_not_match`
- `Is`: `to_be` and `is_`
- `Contains`: `to_contain` and `contains`
- `NotContains`: `to_not_contain` and `does_not_contain`
- `ContainsOnly`: `to_contain_only` and `contains_only`
- `NotContainsOnly`: `to_not_contain_only` and `does_not_contain_only`
- `Length`: `to_have_length` and `has_length`
- `NotLength`: `to_not_have_length` and `does_not_have_length`

CHAPTER 4

Validators

All of the validators in `verify` are callables that can be used in two contexts:

1. By themselves as in `Equal(a, b)` which will raise an `AssertionError` if false.
2. In combination with `expect` as in `expect(a, Equal(b))` which could also raise an `AssertionError`.

The available validators are:

Validator	Description
`Truthy`	Assert that `bool(a)`.
`Falsy`	Assert that `not bool(a)`.
`Not`	Assert that a callable doesn't raise an `AssertionError`.
`Predicate`	Assert that `predicate(a)`.
`All`	Assert that all of the list of predicates evaluate `a` as truthy.
`NotAll`	Assert `not All`.
`Any`	Assert that any of the list of predicates evaluate `a` as truthy.
`NotAny`	Assert `not Any`.
`Equal`	Assert that `a == b`.
`NotEqual`	Assert `not Equal`.
`Match`	Assert that `a` matches regular expression `b`.
`NotMatch`	Assert `not Match`.
`Is`	Assert that `a is b`.
`IsNot`	Assert `not Is`.
`IsTrue`	Assert that `a is True`.
`IsNotTrue`	Assert `not IsTrue`.
`IsFalse`	Assert that `a is False`.
`IsNotFalse`	Assert `not IsFalse`.
`IsNone`	Assert that `a is None`.
`IsNotNone`	Assert `not IsNone`.
`Type`	Assert that `isinstance(a, b)`.
`NotType`	Assert `not Type`.
`Boolean`	Assert that `isinstance(a, bool)`.
`NotBoolean`	Assert `not Boolean`.
`String`	Assert that `isinstance(a, (str, unicode))`.
`NotString`	Assert `not String`.
`Dict`	Assert that `isinstance(a, dict)`.
`NotDict`	Assert `not Dict`.
`List`	Assert that `isinstance(a, list)`.
`NotList`	Assert `not List`.

Continued on next page

Table 4.1 – continued from previous page

Validator	Description
`Tuple`	Assert that `isinstance(a, tuple)`.
`NotTuple`	Assert `not Tuple`.
`Date`	Assert that `isinstance(a, datetime.date)`.
`NotDate`	Assert `not Date`.
`DateString`	Assert that `a` matches the datetime format string `b`.
`NotDateString`	Assert `not DateString`.
`Int`	Assert that `isinstance(a, int)`.
`NotInt`	Assert `not Int`.
`Float`	Assert that `isinstance(a, float)`.
`NotFloat`	Assert `not Float`.
`Number`	Assert that `isinstance(a, (int, float, Decimal, long))`.
`NotNumber`	Assert `not Number`.
`In`	Assert that `a in b`.
`NotIn`	Assert `not In`.
`Contains`	Assert that `b in a`.
`NotContains`	Assert `not Contains`.
`ContainsOnly`	Assert that values from `b` are the only ones contained in `a`.
`NotContainsOnly`	Assert `not ContainsOnly`.
`Subset`	Assert that `a` is a subset of `b`.
`NotSubset`	Assert `not Subset`.
`Superset`	Assert that `a` is a superset of `b`.
`NotSuperset`	Assert `not Superset`.
`Unique`	Assert that `a` contains unique items.
`NotUnique`	Assert `not Unique`.
`Length`	Assert that `b <= len(a) <= c`.
`NotLength`	Assert that `not Length`.
`Greater/GreaterThan`	Assert that `a > b`.
`GreaterEqual/GreaterOrEqual`	Assert that `a >= b`.
`Less/LessThan`	Assert that `a < b`.
`LessEqual/LessOrEqual`	Assert that `a <= b`.
`Between`	Assert that `b <= a <= c`.
`NotBetween`	Assert `not Between`.
`Positive`	Assert that `a > 0`.
`Negative`	Assert that `a < 0`.
`Even`	Assert that `a % 2 == 0`.
`Odd`	Assert that `a % 2 != 1`.
`Monotone`	Assert that `a` is monotonic with respect to `b()`.
`Increasing`	Assert that `a` is monotonically increasing.
`StrictlyIncreasing`	Assert that `a` is strictly increasing.
`Decreasing`	Assert that `a` is monotonically decreasing.
`StrictlyDecreasing`	Assert that `a` is strictly decreasing.

For more details, please see the full documentation at http://verify.readthedocs.org.

CHAPTER 5

Guide

Installation

Verify requires Python >= 2.7 or >= 3.3.

To install from PyPI:

```
pip install verify
```

API Reference

The verify module is composed of various assertion callables (in this case, callable classes) that can be called in two contexts:

1. By themselves as in `Equal(a, b)` which will raise an `AssertionError` if `a` does not equal `b`.
2. In combination with `expect()` as in `expect(a, Equal(b))` which could also raise an `AssertionError`.

Thus, for all assertion classes below, the *value* argument defaults to `NotSet` which is a custom singleton to indicate that nothing was passed in for *value*. Whether *value* is set or `NotSet` is used to indicate which context the assertion class is being used. Whenever *value* is set, the *comparable* is swapped with *value* (internally inside the class' `__init__` method). This allows the assertion to be used in the two contexts above.

This module's main focus is on testing, which is why all assertions raise an `AssertionError` on failure. Therefore, all assertion classes function similarly:

- If the evaluation of *value* with *comparable* returns `False`, then an `AssertionError` is raised with a custom message.
- If the evaluation of *value* with *comparable* returns `True` and the class was only created (e.g. `Equal(a, b)`), then nothing is raised or returned (obviously, since all we did was create a class instance).
- If the evaluation of *value* with *comparable* returns `True` and the class was called (e.g. `expect(a, Equal(b))` or `Equal(b)(a)`), then `True` is returned from the class call.

There are two general types of assertions within this module:

1. Assertions that evaulate a single object: *value*. Referred to here as a plain assertion.
2. Assertions that evaulate two objects: *value* and *comparable*. Referred to here as a comparator assertion.

When using plain assertions with `expect()`, you can pass the bare assertion or initialize it.

```
>>> expect(True, Truthy)
<expect(True)>
>>> expect(True, Truthy())
<expect(True)>
```

When using any of the assertions, inserting `assert` in front is optional as each assertion will raise if the evaluation is false. However, having that `assert` in front may be aesthetically appealing to you, but keep in mind that any assert message included will not be shown since the assertion error will occur within the class itself and raised with it's own custom error message.

```
>>> Truthy(True)
<Truthy()>
>>> assert Truthy(True)
```

```
# Both of these would raise an assertion error.
>>> Falsy(True)
Traceback (most recent call last):
...
AssertionError: True is not falsy

>>> assert Falsy(True)
Traceback (most recent call last):
...
AssertionError: True is not falsy

# But assert messages will not make it to the traceback.
>>> assert Falsy(True), 'this message will not be shown'
Traceback (most recent call last):
...
AssertionError: True is not falsy
```

Assertion Runner

The *expect* class is basically an assertion runner that takes an input *value* and passes it through any number of assertions or predicate functions. If all assertions pass **and** return truthy, then all is well and `True` is returned. Otherwise, either one of the assertion functions will raise an `AssertionError` or no exceptiosn were raised but at least one of the functions returned a non-truthy value which means that `expect()` will return `False`.

The *expect* has alias in the same module under name of `ensure`, so you can use both of these names according to your needs.

class `verify.runners.`**`expect`**(*value*, **assertions*)

Pass *value* through a set of assertable functions.

There are two styles for invoking `expect`:

1. Pass *value* and all *assertions* as arguments to the `__init__` method of `expect`.
2. Pass *value* to the `__init__` method of `expect` and invoke assertions via method chaining.

Examples

Passing *value* and *assertions* to `expect.__init__`:

```
>>> from verify import *
>>> expect(5, Truthy(), Greater(4))
<expect(5)>
```

```
>>> expect(5, Falsy())
Traceback (most recent call last):
...
AssertionError...
```

Using method chaining:

```
>>> expect(5).Truthy().Greater(4)
<expect(5)>
>>> expect(5).Falsy()
Traceback (most recent call last):
...
AssertionError...
```

Parameters

- **value** (*mixed*) – Value to test.
- ***assertions** (*callable, optional*) – Callable objects that accept *value* as its first argument. It's expected that these callables assert something.

Returns Allows for method assertion chaining.

Return type self

Raises `AssertionError` – If the evaluation of all assertions returns `False`.

Aliases:

- `ensure`

New in version 0.0.1.

Changed in version 0.1.0: Rename from `Expect` to `expect` and change implementation from a class to a function. Passed in *value* is no longer called if it's a callable. Return `True` if all assertions pass.

Changed in version 0.6.0: Re-implement as class. Support method chaining of assertion classes. Wrap assertions that are not derived from Assertion in *Predicate* for consistent behavior from external assertion functions.

__getattr__(*attr*)

Invoke assertions via attribute access. All *verify* assertions are available.

Assertions

For all assertion classes, the *value* argument is optional, but when provided the assertion will be evaluated immediately. When passing both the *value* and *comparable* arguments, be sure that *value* comes first even though *comparable* is listed as the first argument. Internally, when both variables are passed in, *value* and *comparable* are swapped in order to support late evaulation, i.e., all of the following are equivalent ways to assert validity:

```
>>> Less(5, 10)
<Less()>
>>> Less(10)(5)
True
>>> expect(5, Less(10))
<expect(5)>
>>> Truthy(5)
<Truthy()>
>>> Truthy()(5)
```

```
True
>>> expect(5, Truthy())
<expect(5)>
```

Below are the various assertion classes that can be used for validation.

Base Classes

Base classes and mixins.

class verify.base.**Assertion**(*value=NotSet*, ***opts*)
Base class for assertions.

If *value* **is not** provided, then assertion isn't executed. This style of usage is used in conjuction with *expect*.

If *value* **is** provided, then assertion is executed immediately. This style of usage is used when making assertions using only the class and not an assertion runner like *expect*.

Keyword Arguments **msg** (*str, optional*) – Override assert message to use when performing assertion.

__call__(**args*, ***opts*)
Execute validation.

Keyword Arguments **msg** (*str, optional*) – Override assert message to use when performing assertion.

Returns True if comparison passes, otherwise, an AssertionError is raised.

Return type bool

Raises AssertionError – If comparison returns False.

format_msg(**args*, ***kargs*)
Return formatted assert message. This is used to generate the assert message during *__call__()*. If no msg keyword argument is provided, then *reason* will be used as the format string. By default, passed in args and kargs along with the classes __dict__ dictionary are given to the format string. In all cases, arg[0] will be the *value* that is being validated.

op = None
Operation to perform to determine whether *value* is valid. **This must be set in subclass**.

reason = ''
Default format string used for assert message.

class verify.base.**Comparator**(*comparable*, *value=NotSet*, ***opts*)
Base class for assertions that compare two values.

class verify.base.**Negate**
Mixin class that negates the results of compare() from the parent class.

verify.base.**NotSet = NotSet**
Singleton to indicate that a keyword argument was not provided.

verify.base.**is_assertion**(*obj*)
Return whether *obj* is either an instance or subclass of *Assertion*.

Logic

Assertions related to logical operations.

class verify.logic.**Truthy**(*value=NotSet*, ***opts*)
Asserts that *value* is truthy.

Aliases:

- to_be_truthy
- is_truthy

New in version 0.0.1.

reason = '{0} is not truthy'

class verify.logic.**Falsy**(*value=NotSet*, ***opts*)
Asserts that *value* is falsy.

Aliases:

- to_be_falsy
- is_falsy

New in version 0.0.1.

reason = '{0} is not falsy'

class verify.logic.**Not**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *comparable* doesn't raise an AssertionError. Can be used to create "opposite" comparators.

Examples

```
>>> from verify import *
>>> expect(5, Not(In([1, 2, 3])))
<expect(5)>
>>> Not(5, In([1, 2, 3]))
<Not()>
>>> Not(In([1, 2, 3]))(5)
True
```

Aliases:

- not_
- does_not
- to_fail
- fails

New in version 0.0.1.

reason = 'The negation of {comparable} should not be true when evaluated with {0}'

class verify.logic.**Predicate**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* evaluated by the predicate *comparable* is True.

Aliases:

- does
- to_pass
- passes

New in version 0.1.0.

Changed in version 0.6.0: Catch `AssertionError` thrown by *comparable* and return `False` as comparison value instead.

`reason` = 'The evaluation of {0} using {comparable} is false'

class `verify.logic.`**`All`**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* evaluates as truthy for **all** predicates in *comparable*.

Aliases:

- `all_`
- `does_all`
- `passes_all`

New in version 0.2.0.

`reason` = '{0} is not true for all {comparable}'

class `verify.logic.`**`NotAll`**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* evaluates as falsy for **all** predicates in *comparable*.

Aliases:

- `to_be_not_all`
- `does_not_all`
- `fails_all`

New in version 0.5.0.

`reason` = '{0} is true for all {comparable}'

class `verify.logic.`**`Any`**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* evaluates as truthy for **any** predicates in *comparable*.

Aliases:

- `any_`
- `does_any`
- `passes_any`

New in version 0.2.0.

`reason` = '{0} is not true for any {comparable}'

class `verify.logic.`**`NotAny`**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* evaluates as falsy for **any** predicates in *comparable*.

Aliases:

- `not_any`
- `does_not_any`
- `fails_any`

New in version 0.5.0.

`reason` = '{0} is true for some {comparable}'

Equality

Assertions related to equality.

class `verify.equality.`**`Equal`**(*comparable*, *value=NotSet*, ***opts*)
Asserts that two values are equal.

Aliases:

- `to_be_equal`
- `is_equal`

New in version 0.0.1.

`reason` = '{0} is not equal to {comparable}'

class `verify.equality.`**`NotEqual`**(*comparable*, *value=NotSet*, ***opts*)
Asserts that two values are not equal.

Aliases:

- `to_not_be_equal`
- `is_not_equal`

New in version 0.5.0.

`reason` = '{0} is equal to {comparable}'

class `verify.equality.`**`Match`**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* matches the regular expression *comparable*.

Parameters

- **`value`** (*mixed, optional*) – Value to compare.
- **`comparable`** (*str|RegExp*) – String or RegExp object used for matching.

Keyword Arguments **flags** (*int, optional*) – Used when compiling regular expression when regular expression is a string. Defaults to `0`.

Aliases:

- `to_match`
- `is_match`
- `matches`

New in version 0.3.0.

`reason` = '{0} does not match the regular expression {comparable}'

class `verify.equality.`**`NotMatch`**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* does not match the regular expression *comparable*.

Aliases:

- `to_not_be_match`
- `is_not_match`
- `not_matches`

New in version 0.5.0.

`reason` = '{0} matches the regular expression {comparable}'

class verify.equality.**Is**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* is *comparable*.

Aliases:

- to_be
- is_

New in version 0.0.1.

reason = '{0} is not {comparable}'

class verify.equality.**IsNot**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* is not *comparable*.

Aliases:

- to_not_be
- is_not

New in version 0.5.0.

reason = '{0} is {comparable}'

class verify.equality.**IsTrue**(*value=NotSet*, ***opts*)
Asserts that *value* is True.

Aliases:

- to_be_true
- is_true

New in version 0.1.0.

reason = '{0} is not True'

class verify.equality.**IsNotTrue**(*value=NotSet*, ***opts*)
Asserts that *value* is not True.

Aliases:

- to_not_be_true
- is_not_true

New in version 0.5.0.

reason = '{0} is True'

class verify.equality.**IsFalse**(*value=NotSet*, ***opts*)
Asserts that *value* is False.

Aliases:

- to_be_false
- is_false

New in version 0.1.0.

reason = '{0} is not False'

class verify.equality.**IsNotFalse**(*value=NotSet*, ***opts*)
Asserts that *value* is not False.

Aliases:

- `to_not_be_false`
- `is_not_false`

New in version 0.5.0.

`reason` = '{0} is False'

class `verify.equality.`**`IsNotNone`**(*value=NotSet*, ***opts*)
Asserts that *value* is not `None`.

Aliases:

- `to_be_not_none`
- `is_not_none`

New in version 0.5.0.

`reason` = '{0} is None'

class `verify.equality.`**`IsNone`**(*value=NotSet*, ***opts*)
Asserts that *value* is `None`.

Aliases:

- `to_be_none`
- `is_none`

New in version 0.0.1.

`reason` = '{0} is not None'

Types

Assertions related to types.

class `verify.types.`**`Type`**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* is an instance of *comparable*.

Aliases:

- `to_be_type`
- `is_type`

New in version 0.0.1.

Changed in version 0.6.0: Renamed from `InstanceOf` to `Type`

`reason` = '{0} is not an instance of {comparable}'

class `verify.types.`**`NotType`**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* is a not an instance of *comparable*.

Aliases:

- `to_be_not_type`
- `is_not_type`

New in version 0.5.0.

Changed in version 0.6.0: Renamed from `NotInstanceOf` to `NotType`

`reason` = '{0} is an instance of {comparable}'

class verify.types.**Boolean**(*value=NotSet*, ***opts*)
Asserts that *value* is a boolean.

Aliases:

- to_be_boolean
- is_boolean

New in version 0.1.0.

reason = '{0} is not a boolean'

class verify.types.**NotBoolean**(*value=NotSet*, ***opts*)
Asserts that *value* is a not a boolean.

Aliases:

- to_be_not_boolean
- is_not_boolean

New in version 0.5.0.

reason = '{0} is a boolean'

class verify.types.**String**(*value=NotSet*, ***opts*)
Asserts that *value* is a string (str or unicode on Python 2).

Aliases:

- to_be_string
- is_string

New in version 0.1.0.

reason = '{0} is not a string'

class verify.types.**NotString**(*value=NotSet*, ***opts*)
Asserts that *value* is a not a string.

Aliases:

- to_be_not_string
- is_not_string

New in version 0.5.0.

reason = '{0} is a string'

class verify.types.**Dict**(*value=NotSet*, ***opts*)
Asserts that *value* is a dictionary.

Aliases:

- to_be_dict
- is_dict

New in version 0.1.0.

reason = '{0} is not a dictionary'

class verify.types.**NotDict**(*value=NotSet*, ***opts*)
Asserts that *value* is a not a dict.

Aliases:

- `to_be_not_dict`
- `is_dict`

New in version 0.5.0.

`reason` = '{0} is a dict'

class `verify.types.`**`List`**(*value=NotSet*, ***opts*)
Asserts that *value* is a list.

Aliases:

- `to_be_list`
- `is_list`

New in version 0.1.0.

`reason` = '{0} is not a list'

class `verify.types.`**`NotList`**(*value=NotSet*, ***opts*)
Asserts that *value* is a not a list.

Aliases:

- `to_be_not_list`
- `is_not_list`

New in version 0.5.0.

`reason` = '{0} is a list'

class `verify.types.`**`Tuple`**(*value=NotSet*, ***opts*)
Asserts that *value* is a tuple.

Aliases:

- `to_be_tuple`
- `is_tuple`

New in version 0.1.0.

`reason` = '{0} is not a tuple'

class `verify.types.`**`NotTuple`**(*value=NotSet*, ***opts*)
Asserts that *value* is a not a tuple.

Aliases:

- `to_be_not_tuple`
- `is_not_tuple`

New in version 0.5.0.

`reason` = '{0} is a tuple'

class `verify.types.`**`Date`**(*value=NotSet*, ***opts*)
Asserts that *value* is an instance of `datetime.date` or `datetime.datetime`.

Aliases:

- `to_be_date`
- `is_date`

New in version 0.3.0.

reason = '{0} is not a date or datetime object'

class verify.types.**NotDate**(*value=NotSet*, ***opts*)
Asserts that *value* is a not a date or datetime object.

Aliases:

- to_be_not_date
- is_not_date

New in version 0.5.0.

reason = '{0} is a date or datetime object'

class verify.types.**DateString**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* is matches the datetime format string *comparable*.

Aliases:

- to_be_date_string
- is_date_string

New in version 0.3.0.

reason = '{0} does not match the datetime format {comparable}'

class verify.types.**NotDateString**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* does not match datetime format string *comparable*.

Aliases:

- to_be_not_date_string
- is_not_date_string

New in version 0.5.0.

reason = '{0} matches the datetime format {comparable}'

class verify.types.**Int**(*value=NotSet*, ***opts*)
Asserts that *value* is an integer.

Aliases:

- to_be_int
- is_int

New in version 0.1.0.

reason = '{0} is not an integer'

class verify.types.**NotInt**(*value=NotSet*, ***opts*)
Asserts that *value* is a not an integer.

Aliases:

- to_be_not_int
- is_not_int

New in version 0.5.0.

reason = '{0} is an integer'

class verify.types.**NotFloat**(*value=NotSet*, ***opts*)
Asserts that *value* is a not a float.

Aliases:

- `to_be_not_float`
- `is_not_float`

New in version 0.5.0.

`reason` = '{0} is a float'

class `verify.types.`**`Float`**(*value=NotSet*, ***opts*)
Asserts that *value* is a float.

Aliases:

- `to_be_float`
- `is_float`

New in version 0.1.0.

`reason` = '{0} is not a float'

class `verify.types.`**`Number`**(*value=NotSet*, ***opts*)
Asserts that *value* is a number.

Objects considered a number are:

- `int`
- `float`
- `decimal.Decimal`
- `long (Python 2)`

Aliases:

- `to_be_number`
- `is_number`

New in version 0.1.0.

`reason` = '{0} is not a number'

class `verify.types.`**`NotNumber`**(*value=NotSet*, ***opts*)
Asserts that *value* is a not a number.

Aliases:

- `to_be_not_number`
- `is_not_number`

New in version 0.1.0.

Changed in version 0.5.0: Renamed from `NaN` to `NotNumber`.

`reason` = '{0} is a number'

Containers

Assertions related to containers/iterables.

class `verify.containers.`**`In`**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* is in *comparable*.

Aliases:

- `to_be_in`
- `is_in`

New in version 0.0.1.

`reason` = '{0} is not in {comparable}'

class `verify.containers.`**`NotIn`**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* is not in *comparable*.

Aliases:

- `to_not_be_in`
- `is_not_in`

New in version 0.5.0.

`reason` = '{0} is in {comparable}'

class `verify.containers.`**`Contains`**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* is an iterable and contains *comparable*.

Aliases:

- `to_contain`
- `contains`

New in version 0.2.0.

`reason` = '{0} does not contain {comparable}'

class `verify.containers.`**`NotContains`**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* does not contain *comparable*.

Aliases:

- `to_not_contain`
- `does_not_contain`

New in version 0.5.0.

`reason` = '{0} contains {comparable}'

class `verify.containers.`**`ContainsOnly`**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* is an iterable and only contains *comparable*.

Aliases:

- `to_contain_only`
- `contains_only`

New in version 0.2.0.

`reason` = '{0} does not only contain values in {comparable}'

class `verify.containers.`**`NotContainsOnly`**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* does not contain only *comparable*.

Aliases:

- `to_not_contain_only`
- `does_not_contain_only`

New in version 0.5.0.

reason = '{0} contains only {comparable}'

class verify.containers.**Subset**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* is a subset of *comparable*. Comparison supports nested `dict`, `list`, and `tuple` objects.

Aliases:

- `to_be_subset`
- `is_subset`

New in version 0.3.0.

reason = '{0} is not a subset of {comparable}'

class verify.containers.**NotSubset**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* is a not a subset of *comparable*.

Aliases:

- `to_not_be_subset`
- `is_not_subset`

New in version 0.5.0.

reason = '{0} is a subset of {comparable}'

class verify.containers.**Superset**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* is a superset of *comparable*. Comparison supports nested `dict`, `list`, and `tuple` objects.

Aliases:

- `to_be_superset`
- `is_superset`

New in version 0.3.0.

reason = '{0} is not a supserset of {comparable}'

class verify.containers.**NotSuperset**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* is a not a superset of *comparable*.

Aliases:

- `to_not_be_superset`
- `is_not_superset`

New in version 0.5.0.

reason = '{0} is a superset of {comparable}'

class verify.containers.**Unique**(*value=NotSet*, ***opts*)
Asserts that *value* contains only unique values. If *value* is a `dict`, then its `values()` will be compared.

Aliases:

- `to_be_unique`
- `is_unique`

New in version 0.3.0.

reason = '{0} contains duplicate items'

class verify.containers.**NotUnique**(*value=NotSet*, ***opts*)
Asserts that *value* is a not a unique.

Aliases:

- to_not_be_unique
- is_not_unique

New in version 0.5.0.

reason = '{0} is unique'

class verify.containers.**Length**(*value=NotSet*, ***opts*)
Asserts that *value* is an iterable with length between *min* and *max* inclusively.

Examples

These will pass:

```
>>> assert Length([1, 2, 3], min=3, max=3)  # 3 <= len(a) <= 3
>>> assert Length([1, 2, 3, 4, 5], min=5, max=6)  # 5 <= len(a) <= 6
>>> assert Length([1, 2, 3], max=6)  # len(a) <= 6
>>> assert Length([1, 2, 3, 4], min=4)  # len(a) >= 4
```

This will fail:

```
>>> Length([1, 2, 4], max=2)  # len(a) <= 2
Traceback (most recent call last):
...
AssertionError...
```

Parameters **value** (*mixed, optional*) – Value to compare.

Keyword Arguments

- **min** (*int, optional*) – Minimum value that *value* must be greater than or equal to.
- **max** (*int, optional*) – Maximum value that *value* must be less than or equal to.

Aliases:

- to_have_length
- has_length

New in version 0.2.0.

Changed in version 0.4.0: Change comparison to function like Between meaning length is compared to min and max values. Allow keyword arguments min and max to be used in place of positional tuple

Changed in version 1.0.0: Removed positional tuple argument and only support min and max keyword arguments.

reason = '{0} does not have length between {min} and {max}'

class verify.containers.**NotLength**(*value=NotSet*, ***opts*)
Asserts that *value* is an iterable with length not between *min* and *max* inclusively.

Aliases:

- to_not_have_length

- does_not_have_length

New in version 1.0.0.

reason = '{0} has length between {min} and {max}'

Numbers

Assertions related to numbers.

class verify.numbers.**Greater**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* is greater than *comparable*.

Aliases:

- GreaterThan
- to_be_greater
- to_be_greater_than
- is_greater
- is_greater_than

New in version 0.0.1.

reason = '{0} is not greater than {comparable}'

verify.numbers.**GreaterThan**
alias of *Greater*

class verify.numbers.**GreaterEqual**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* is greater than or equal to *comparable*.

Aliases:

- GreaterThanEqual
- to_be_greater_equal
- to_be_greater_or_equal
- is_greater_equal
- is_greater_or_equal

New in version 0.0.1.

reason = '{0} is not greater than or equal to {comparable}'

verify.numbers.**GreaterOrEqual**
alias of *GreaterEqual*

class verify.numbers.**Less**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* is less than *comparable*.

Aliases:

- LessThan
- to_be_less
- to_be_less_than
- is_less
- is_less_than

New in version 0.0.1.

reason = '{0} is not less than {comparable}'

verify.numbers.**LessThan**
alias of *Less*

class verify.numbers.**LessEqual**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* is less than or equal to *comparable*.

Aliases:

- LessThanEqual
- to_be_less_equal
- to_be_less_or_equal
- is_less_equal
- is_less_or_equal

New in version 0.0.1.

reason = '{0} is not less than or equal to {comparable}'

verify.numbers.**LessOrEqual**
alias of *LessEqual*

class verify.numbers.**Between**(*value=NotSet*, ***opts*)
Asserts that *value* is between *min* and *max* inclusively.

Examples

These will pass:

```
>>> assert Between(5, min=4, max=6)  # 4 <= 5 <= 6
>>> assert Between(5, min=5, max=6)  # 5 <= 5 <= 6
>>> assert Between(5, max=6)  # 5 <= 6
>>> assert Between(5, min=4)  # 5 >= 4
```

This will fail:

```
>>> Between(5, max=4)  # 5 <= 4
Traceback (most recent call last):
...
AssertionError...
```

Parameters **value** (*mixed, optional*) – Value to compare.

Keyword Arguments

- **min** (*int, optional*) – Minimum value that *value* must be greater than or equal to.
- **max** (*int, optional*) – Maximum value that *value* must be less than or equal to.

Aliases:

- to_be_between
- is_between

New in version 0.2.0.

Changed in version 0.4.0: Allow keyword arguments `min` and `max` to be used in place of positional tuple.

Changed in version 1.0.0: Removed positional tuple argument and only support `min` and `max` keyword arguments.

`reason` = '{0} is not between {min} and {max}'

class `verify.numbers.`**`NotBetween`**(*value=NotSet*, ***opts*)
Asserts that *value* is not between *min* and *max* inclusively.

Aliases:

- `to_not_be_between`
- `is_not_between`

New in version 0.5.0.

`reason` = '{0} is between {min} and {max}'

class `verify.numbers.`**`Positive`**(*value=NotSet*, ***opts*)
Asserts that *value* is a positive number.

Aliases:

- `to_be_positive`
- `is_positive`

New in version 0.3.0.

`reason` = '{0} is not a positive number'

class `verify.numbers.`**`Negative`**(*value=NotSet*, ***opts*)
Asserts that *value* is a negative number.

Aliases:

- `to_be_negative`
- `is_negative`

New in version 0.3.0.

`reason` = '{0} is not a negative number'

class `verify.numbers.`**`Even`**(*value=NotSet*, ***opts*)
Asserts that *value* is an even number.

Aliases:

- `to_be_even`
- `is_even`

New in version 0.3.0.

`reason` = '{0} is not an even number'

class `verify.numbers.`**`Odd`**(*value=NotSet*, ***opts*)
Asserts that *value* is an odd number.

Aliases:

- `to_be_odd`
- `is_odd`

New in version 0.3.0.

reason = '{0} is not an odd number'

class verify.numbers.**Monotone**(*comparable*, *value=NotSet*, ***opts*)
Asserts that *value* is a monotonic with respect to *comparable*.

Aliases:

- to_be_monotone
- is_monotone

New in version 0.3.0.

reason = '{0} is not monotonic as evaluated by {comparable}'

class verify.numbers.**Increasing**(*value=NotSet*, ***opts*)
Asserts that *value* is monotonically increasing.

Aliases:

- to_be_increasing
- is_increasing

New in version 0.3.0.

reason = '{0} is not monotonically increasing'

class verify.numbers.**StrictlyIncreasing**(*value=NotSet*, ***opts*)
Asserts that *value* is strictly increasing.

Aliases:

- to_be_strictly_increasing
- is_strictly_increasing

New in version 0.3.0.

reason = '{0} is not strictly increasing'

class verify.numbers.**Decreasing**(*value=NotSet*, ***opts*)
Asserts that *value* is monotonically decreasing.

Aliases:

- to_be_decreasing
- is_decreasing

New in version 0.3.0.

reason = '{0} is not monotonically decreasing'

class verify.numbers.**StrictlyDecreasing**(*value=NotSet*, ***opts*)
Asserts that *value* is strictly decreasing.

Aliases:

- to_be_strictly_decreasing
- is_strictly_decreasing

New in version 0.3.0.

reason = '{0} is not strictly decreasing'

CHAPTER 6

Project Info

License

Versioning

This project follows Semantic Versioning with the following caveats:

- Only the public API (i.e. the objects imported into the `verify` module) will maintain backwards compatibility between MINOR version bumps.
- Objects within any other parts of the library are not guaranteed to not break between MINOR version bumps.

With that in mind, it is recommended to only use or import objects from the main module, `verify`.

Changelog

v1.1.1 (2017-05-09)

- Fix compatibility with pydash v4.

v1.1.0 (2015-07-23)

- Add `ensure` as alias of `expect`.
- Add `to_be_*` and `is_*` aliases for all assertions.

v1.0.0 (2015-05-15)

- Add `NotLength`.
- Make assertions accept an optional argument, `msg`, that overrides the default assert message on a per call basis.
- Make `Between` and `Length` only accept keyword arguments `min` and `max`. (**breaking change**)

v0.6.0 (2015-05-14)

- Make `expect` into a class and support method chaining of assertions. Original usage is still supported.
- Make `expect` wrap external predicate functions with `Predicate` for evaluation. (**breaking change**)
- Make `Predicate` catch `AssertionError` thrown by *comparable* and return `False`. (**breaking change**)
- Make `Predicate` treat a *comparable* that returns `None` as passing. (**breaking change**)
- Rename `InstanceOf` and `NotInstanceOf` to `Type` and `NotType`. (**breaking change**)

v0.5.0 (2015-05-12)

- Add `NotEqual`.
- Add `NotMatch`.
- Add `NotBetween`.
- Add `IsNot`.
- Add `IsNotTrue`.
- Add `IsNotFalse`.
- Add `IsNotNone`.
- Add `NotAll`.
- Add `NotAny`.
- Add `NotIn`.
- Add `NotContains`.
- Add `NotContainsOnly`.
- Add `NotSubset`.
- Add `NotSuperset`.
- Add `NotUnique`.
- Add `NotInstanceOf`.
- Add `NotBoolean`.
- Add `NotString`.

- Add `NotDict`.
- Add `NotList`.
- Add `NotTuple`.
- Add `NotDate`.
- Add `NotDateString`.
- Add `NotInt`.
- Add `NotFloat`.
- Rename `NaN` to `NotNumber`. (**breaking change**)

v0.4.0 (2015-05-12)

- Make `Between` accept keyword arguments for `min` and `max`.
- Make `Length` function like `Between` and allow comparison over range of lengths. If a single comparable value is passed in, then comparison uses the value as a max length. Previously, a single comparable value performed an equality check for length. (**breaking change**)
- Make `Match` accept keyword argument `flags` for use with string based regular expression.

v0.3.0 (2015-05-11)

- Add `Match`.
- Add `Subset`.
- Add `Superset`.
- Add `Unique`.
- Add `Date`.
- Add `DateString`.
- Add `Positive`.
- Add `Negative`.
- Add `Even`.
- Add `Odd`.
- Add `Monotone`.
- Add `Increasing`.
- Add `StrictlyIncreasing`.
- Add `Decreasing`.
- Add `StrictlyDecreasing`.

v0.2.0 (2015-05-11)

- Add `All`.
- Add `Any`.
- Add `Between`.
- Add `Contains`.
- Add `ContainsOnly`.
- Add `Length`.
- Make `Not` compatible with bare predicate functions by return the evaluation of the *comparable*.

v0.1.1 (2015-05-08)

- Make `expect` include an assertion message on failure. Without it, a cryptic `NameError` is thrown when a plain predicate function fails due to a generator being used in the `all()` call.

v0.1.0 (2015-05-08)

- Add `Boolean`.
- Add `Dict`.
- Add `Float`.
- Add `Int`.
- Add `IsTrue`.
- Add `IsFalse`.
- Add `List`.
- Add `NaN`.
- Add `Number`.
- Add `Predicate`.
- Add `String`.
- Add `Tuple`.
- Rename `Except` to `except`. (**breaking change**)
- Make `except` **not** call *value* if it's callable. (**breaking change**)
- Make `except` return `True` if all assertions pass.

v0.0.1 (2015-05-07)

- First release.

Authors

Lead

- Derrick Gilland, dgilland@gmail.com, dgilland@github

Contributors

- Szczepan Cieślik, szczepan.cieslik@gmail.com, beregond@github

How to Contribute

Overview

1. Fork the repo.
2. Build development environment run tests to ensure a clean, working slate.
3. Improve/fix the code.
4. Add test cases if new functionality introduced or bug fixed (100% test coverage).
5. Ensure tests pass.
6. Add yourself to `AUTHORS.rst`.
7. Push to your fork and submit a pull request to the `develop` branch.

Guidelines

Some simple guidelines to follow when contributing code:

- Adhere to PEP8.
- Clean, well documented code.
- All tests must pass.
- 100% test coverage.

Branching

There are two main development branches: `master` and `develop`. `master` represents the currently released version while `develop` is the latest development work. When submitting a pull request, be sure to submit to `develop`. The originating branch you submit from can be any name though.

Continuous Integration

Integration testing is provided by Travis-CI at https://travis-ci.org/dgilland/verify.

Test coverage reporting is provided by Coveralls at https://coveralls.io/r/dgilland/verify.

Project CLI

Some useful CLI commands when working on the project are below. **NOTE:** All commands are run from the root of the project and require `make`.

make build

Run the `clean` and `install` commands.

```
make build
```

make install

Install Python dependencies into virtualenv located at `env/`.

```
make install
```

make clean

Remove build/test related temporary files like `env/`, `.tox`, `.coverage`, and `__pycache__`.

```
make clean
```

make test

Run unittests under the virtualenv's default Python version. Does not test all support Python versions. To test all supported versions, see *make test-full*.

```
make test
```

make test-full

Run unittest and linting for all supported Python versions. **NOTE:** This will fail if you do not have all Python versions installed on your system. If you are on an Ubuntu based system, the Dead Snakes PPA is a good resource for easily installing multiple Python versions. If for whatever reason you're unable to have all Python versions on your development machine, note that Travis-CI will run full integration tests on all pull requests.

```
make test-full
```

make lint

Run `make pylint` and `make pep8` commands.

```
make lint
```

make pylint

Run `pylint` compliance check on code base.

```
make pylint
```

make pep8

Run PEP8 compliance check on code base.

```
make pep8
```

make docs

Build documentation to `docs/_build/`.

```
make docs
```

CHAPTER 7

Indices and Tables

- genindex
- modindex
- search

V

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V