Ensure Documentation

Release 0.0.1

Andrey Kislyuk

Jun 01, 2017

Contents

ensure is a set of simple assertion helpers that let you write more expressive, literate, concise, and readable Pythonic code for validating conditions. It's inspired by should.js, expect.js, and builds on top of the unittest/JUnit assert helpers.

If you use Python 3, you can use *ensure* to enforce your **function signature annotations**: see PEP 3107 and the @ensure_annotations decorator below.

Because *ensure* is fast, is a standalone library (not part of a test framework), doesn't monkey-patch anything or use DSLs, and doesn't use the assert statement (which is liable to be turned off with the `-O` flag), it can be used to validate conditions in production code, not just for testing (though it certainly works as a BDD test utility library).

Aside from better looking code, a big reason to use *ensure* is that it provides more consistent, readable, and informative error messages when things go wrong. See Motivation and Goals for more.

CHAPTER 1

Installation

```
pip install ensure
```

CHAPTER 2

Synopsis

```
from ensure import ensure

ensure(1).is_an(int)
ensure({1: {2: 3}}).equals({1: {2: 3}}).also.contains(1)
ensure({1: "a"}).has_key(1).whose_value.has_length(1)
ensure.each_of([{1: 2}, {3: 4}]).is_a(dict).of(int).to(int)
ensure(int).called_with("1100101", base=2).returns(101)
ensure(dict).called_with(1, 2).raises(TypeError)
check(1).is_a(float).or_raise(Exception, "An error happened: {msg}. See http://
↪example.com for more information.")
```

In Python 3:

```
from ensure import ensure_annotations

@ensure_annotations
def f(x: int, y: float) -> float:
    return x+y
```

See **More examples** below.

Notes

The `ensure` module exports the `Ensure` class and its convenience instance `ensure`. Instances of the class are callable, and the call will reset the contents that the instance is inspecting, so you can reuse it for many checks (as seen above).

The class raises `EnsureError` (a subclass of `AssertionError`) by default.

There are several ways to **chain clauses**, depending on the grammatical context: `.also`, `.which`, and `.whose_value` are available per examples below.

Raising custom exceptions

You can pass a callable or exception class as the `error_factory` keyword argument to `Ensure()`, or you can use the `Check` class or its convenience instance `check()`. This class behaves like `Ensure`, but does not raise errors immediately. It saves them and chains the methods `otherwise()`, `or_raise()` and `or_call()` to the end of the clauses.

```
from ensure import check

check("w00t").is_an(int).or_raise(Exception)
check(1).is_a(float).or_raise(Exception, "An error happened: {msg}. See http://
↪example.com for more information.")
check("w00t").is_an(int).or_raise(MyException, 1, 2, x=3, y=4)
```

```
def build_fancy_exception(original_exception):
    return MyException(original_exception)

check("w00t").is_an(int).otherwise(build_fancy_exception)
check("w00t").is_an(int).or_call(build_fancy_exception, *args, **kwargs)
```

CHAPTER 3

More examples

```
ensure({1: {2: 3}}).is_not_equal_to({1: {2: 4}})
ensure(True).does_not_equal(False)
ensure(1).is_in(range(10))
ensure(True).is_a(bool)
ensure(True).is_(True)
ensure(True).is_not(False)
```

```
ensure(["train", "boat"]).contains_one_of(["train"])
ensure(range(8)).contains(5)
ensure(["spam"]).contains_none_of(["eggs", "ham"])
ensure("abcdef").contains_some_of("abcxyz")
ensure("abcdef").contains_one_or_more_of("abcxyz")
ensure("abcdef").contains_all_of("acf")
ensure("abcd").contains_only("dcba")
ensure("abc").does_not_contain("xyz")
ensure([1, 2, 3]).contains_no(float)
ensure(1).is_in(range(10))
ensure("z").is_not_in("abc")
ensure(None).is_not_in([])
ensure(dict).has_attribute('__contains__').which.is_callable()
ensure({1: "a", 2: "b", 3: "c"}).has_keys([1, 2])
ensure({1: "a", 2: "b"}).has_only_keys([1, 2])
```

```
ensure(1).is_true()
ensure(0).is_false()
ensure(None).is_none()
ensure(1).is_not_none()
ensure("").is_empty()
ensure([1, 2]).is_nonempty().also.has_length(2)
ensure(1.1).is_a(float).which.equals(1.10)
ensure(KeyError()).is_an(Exception)
ensure({x: str(x) for x in range(5)}).is_a_nonempty(dict).of(int).to(str)
ensure({}).is_an_empty(dict)
ensure(None).is_not_a(list)
```

```
import re
ensure("abc").matches("A", flags=re.IGNORECASE)
ensure([1, 2, 3]).is_an_iterable_of(int)
ensure([1, 2, 3]).is_a_list_of(int)
ensure({1, 2, 3}).is_a_set_of(int)
ensure({1: 2, 3: 4}).is_a_mapping_of(int).to(int)
ensure({1: 2, 3: 4}).is_a_dict_of(int).to(int)
ensure({1: 2, 3: 4}).is_a(dict).of(int).to(int)
ensure(10**100).is_numeric()
ensure(lambda: 1).is_callable()
ensure("abc").has_length(3)
ensure("abc").has_length(min=3, max=8)
ensure(1).is_greater_than(0)
ensure(1).exceeds(0)
ensure(0).is_less_than(1)
ensure(1).is_greater_than_or_equal_to(1)
ensure(0).is_less_than_or_equal_to(0)
ensure(1).is_positive()
ensure(1.1).is_a_positive(float)
ensure(-1).is_negative()
ensure(-1).is_a_negative(int)
ensure(0).is_nonnegative()
ensure(0).is_a_nonnegative(int)
```

```
ensure("{x} {y}".format).called_with(x=1, y=2).equals("1 2")
ensure(int).called_with("1100101", base=2).returns(101)
ensure("{x} {y}".format).with_args(x=1, y=2).is_a(str)
with ensure().raises(ZeroDivisionError):
    1/0
with ensure().raises_regex(NameError, "'w00t' is not defined"):
    w00t
```

See complete API documentation.

Enforcing function annotations

Use the `@ensure_annotations` decorator to enforce function signature annotations:

```
from ensure import ensure_annotations

@ensure_annotations
def f(x: int, y: float) -> float:
    return x+y

f(1, 2.3)
```

```
>>> 3.3
```

```
f(1, 2)
```

```
>>> ensure.EnsureError: Argument y to <function f at 0x109b7c710> does not match␣
↪annotation type <class 'float'>
```

Compare this runtime type checking to compile-time checking in Mypy and type hinting in PEP 484/Python 3.5+.

Motivation and goals

Many BDD assertion libraries suffer from an excess of magic, or end up having to construct statements that don't parse as English easily. *ensure* is deliberately kept simple to avoid succumbing to either issue. The source is easy to read and extend.

Work remains to make error messages raised by *ensure* even more readable, informative, and consistent. Going forward, ability to introspect exceptions to extract structured error information will be a major development focus. You will be in control of how much information is presented in each error, which context it's thrown from, and what introspection capabilities the exception object will have.

The original use case for *ensure* is as an I/O validation helper for API endpoints, where the client needs to be sent a very clear message about what went wrong, some structured information (such as an HTTP error code and machine-readable reference to a failing element) may need to be added, and some information may need to be hidden from the client. To further improve on that, we will work on better error translation, marshalling, message formatting, and schema validation helpers.

CHAPTER 4

Authors

- Andrey Kislyuk
- Harrison Metzger

CHAPTER 5

Links

- Project home page (GitHub)
- Documentation (Read the Docs)
- Package distribution (PyPI)

Bugs

Please report bugs, issues, feature requests, etc. on GitHub.

CHAPTER 6

License

Licensed under the terms of the Apache License, Version 2.0.

CHAPTER 7

API documentation

class ensure.**Ensure**(*subject=None*, *error_factory=<class 'ensure.EnsureError'>*, *catch=<type 'exceptions.Exception'>*)

Constructs a root-level inspector, which can perform a variety of checks (*predicates*) on subjects passed to it. If the checks do not pass, by default `EnsureError` is raised. This can be configured by passing the `error_factory` keyword to the constructor.

Subjects can be passed to the inspector at construction time or by calling the resulting object (each call resets the subject):

```
Ensure(1).is_an(int)

e = Ensure()
e(1).is_an(int)
```

Some predicates return child inspectors which can be chained into a series of predicates, for example:

```
ensure({1: {2: "a"}}).has_key(1).whose_value.is_a(dict).of(int).to(str)
```

called_with(**args*, ***kwargs*)

Before evaluating subsequent predicates, calls `subject` with given arguments (but unlike a direct call, catches and transforms any exceptions that arise during the call).

contains(*element*)

Ensures `subject` contains *other*.

contains_all_of(*elements*)

Ensures `subject` contains all of *elements*, which must be an iterable.

contains_no(*prototype*)

Ensures no item of `subject` is of class *prototype*.

contains_none_of(*elements*)

Ensures `subject` contains none of *elements*, which must be an iterable.

contains_one_of(*elements*)

Ensures `subject` contains exactly one of *elements*, which must be an iterable.

contains_one_or_more_of(*elements*)
Ensures `subject` contains at least one of *elements*, which must be an iterable.

contains_only(*elements*)
Ensures `subject` contains all of *elements*, which must be an iterable, and no other items.

contains_some_of(*elements*)
Ensures `subject` contains at least one of *elements*, which must be an iterable.

does_not_contain(*element*)
Ensures `subject` does not contain *element*.

does_not_equal(*other*)
Ensures `subject` is not equal to *other*.

classmethod each_of(*iterable*)
Applies subsequent predicate(s) to all items in *iterable*.

equals(*other*)
Ensures `subject` is equal to *other*.

exceeds(*other*)
Ensures `subject` is greater than *other*.

has_attribute(*attr*)
Ensures `subject` has an attribute *attr*.

has_key(*key*)
Ensures `subject` is a `collections.Mapping` and contains *key*.

has_keys(*keys*)
Ensures `subject` is a `collections.Mapping` and contains *keys*, which must be an iterable.

has_length(*length=None*, *min=None*, *max=None*)
Ensures `subject` has length *length* (if given), length at least *min* (if given), and length at most *max* (if given).

has_only_keys(*keys*)
Ensures `subject` is a `collections.Mapping` and contains *keys*, and no other keys.

hasattr(*attr*)
Ensures `subject` has an attribute *attr*.

is_(*other*)
Ensures `subject` is *other* (object identity check).

is_a(*prototype*)
Ensures `subject` is an object of class *prototype*.

is_a_dict_of(*prototype*)
Ensures `subject` is a `dict` containing only objects of class *prototype*.

is_a_list_of(*prototype*)
Ensures `subject` is a `list` containing only objects of class *prototype*.

is_a_mapping_of(*prototype*)
Ensures `subject` is a `collections.Mapping` containing only objects of class *prototype*.

is_a_negative(*prototype*)
Ensures `subject` is less than 0 and is an object of class *prototype*.

is_a_nonempty(*prototype*)
Ensures `subject` is an object of class *prototype* and has non-zero length.

is_a_nonnegative(*prototype*)
Ensures `subject` is greater than or equal to 0 and is an object of class *prototype*.

is_a_positive(*prototype*)
Ensures `subject` is greater than 0 and is an object of class *prototype*.

is_a_set_of(*prototype*)
Ensures `subject` is a `set` containing only objects of class *prototype*.

is_an(*prototype*)
Ensures `subject` is an object of class *prototype*.

is_an_empty(*prototype*)
Ensures `subject` is an object of class *prototype* and has zero length.

is_an_instance_of(*prototype*)
Ensures `subject` is an object of class *prototype*.

is_an_iterable_of(*prototype*)
Ensures `subject` is an iterable containing only objects of class *prototype*.

is_callable()
Ensures `subject` is a callable.

is_empty()
Ensures `subject` has length zero.

is_false()
Ensures `subject` is `False`.

is_greater_than(*other*)
Ensures `subject` is greater than *other*.

is_greater_than_or_equal_to(*other*)
Ensures `subject` is greater than or equal to *other*.

is_in(*iterable*)
Ensures `subject` is contained in *iterable*.

is_less_than(*other*)
Ensures `subject` is less than *other*.

is_less_than_or_equal_to(*other*)
Ensures `subject` is less than or equal to *other*.

is_negative()
Ensures `subject` is less than 0.

is_none()
Ensures `subject` is `None`.

is_none_or
Ensures `subject` is either `None`, or satisfies subsequent (chained) conditions:

```
Ensure(None).is_none_or.is_an(int)
```

is_nonempty()
Ensures `subject` has non-zero length.

is_nonnegative()
Ensures `subject` is greater than or equal to 0.

is_not(*other*)
Ensures `subject` is not *other* (object identity check).

is_not_a(*prototype*)
Ensures `subject` is not an object of class *prototype*.

is_not_equal_to(*other*)
Ensures `subject` is not equal to *other*.

is_not_in(*iterable*)
Ensures `subject` is not contained in *iterable*.

is_not_none()
Ensures `subject` is not `None`.

is_numeric()
Ensures `subject` is an int, float, or long.

is_positive()
Ensures `subject` is greater than 0.

is_true()
Ensures `subject` is `True`.

matches(*pattern*, *flags=0*)
Ensures `subject` matches regular expression *pattern*.

not_a(*prototype*)
Ensures `subject` is not an object of class *prototype*.

not_in(*iterable*)
Ensures `subject` is not contained in *iterable*.

raises(*expected_exception*)
Ensures preceding predicates (specifically, *`called_with()`*) result in *expected_exception* being raised.

raises_regex(*expected_exception*, *expected_regexp*)
Ensures preceding predicates (specifically, *`called_with()`*) result in *expected_exception* being raised, and the string representation of *expected_exception* must match regular expression *expected_regexp*.

with_args(**args*, ***kwargs*)
Before evaluating subsequent predicates, calls `subject` with given arguments (but unlike a direct call, catches and transforms any exceptions that arise during the call).

class `ensure.`**Check**(**args*, ***kwargs*)
Like Ensure, but if a check fails, saves the error instead of raising it immediately. The error can then be acted upon using *`or_raise()`* or *`or_call()`*.

`.each_of()` is not supported by the **Check** inspector; all other methods are supported.

or_call(*_callable*, **args*, ***kwargs*)
Calls **_callable** with supplied args and kwargs if a predicate fails.

or_raise(*error_factory*, *message=None*, **args*, ***kwargs*)
Raises an exception produced by **error_factory** if a predicate fails.

Parameters

- **error_factory** – Class or callable (e.g. `Exception`) which will be invoked to produce the resulting exception. You can define a custom callable here; it will be given the underlying predicate's exception (AssertError) as the first argument, followed by any arguments passed to `or_raise`.
- **message** – String to be formatted and passed as the first argument to *error_factory*. If this is given, subsequent arguments passed to `or_raise` will be used to format contents

of the string, and will not be passed to `error_factory`. The keyword argument `error` will be set to the underlying predicate's exception.

otherwise (*error_factory*, *message=None*, **args*, ***kwargs*)
Raises an exception produced by **error_factory** if a predicate fails.

Parameters

- **error_factory** – Class or callable (e.g. `Exception`) which will be invoked to produce the resulting exception. You can define a custom callable here; it will be given the underlying predicate's exception (AssertError) as the first argument, followed by any arguments passed to `or_raise`.
- **message** – String to be formatted and passed as the first argument to *error_factory*. If this is given, subsequent arguments passed to `or_raise` will be used to format contents of the string, and will not be passed to `error_factory`. The keyword argument `error` will be set to the underlying predicate's exception.

ensure.**ensure_annotations**(*f*)
Decorator to be used on functions with annotations. Runs type checks to enforce annotations. Raises `EnsureError` if any argument passed to *f* is not of the type specified by the annotation. Also raises `EnsureError` if the return value of *f* is not of the type specified by the annotation. Examples:

```
from ensure import ensure_annotations

@ensure_annotations
def f(x: int, y: float) -> float:
    return x+y

print(f(1, y=2.2))

>>> 3.2

print(f(1, y=2))

>>> ensure.EnsureError: Argument y to <function f at 0x109b7c710> does not match␣
↪annotation type <class 'float'>
```

ensure.**ensure_annotations**(*f*)
Decorator to be used on functions with annotations. Runs type checks to enforce annotations. Raises `EnsureError` if any argument passed to *f* is not of the type specified by the annotation. Also raises `EnsureError` if the return value of *f* is not of the type specified by the annotation. Examples:

```
from ensure import ensure_annotations

@ensure_annotations
def f(x: int, y: float) -> float:
    return x+y

print(f(1, y=2.2))

>>> 3.2

print(f(1, y=2))

>>> ensure.EnsureError: Argument y to <function f at 0x109b7c710> does not match␣
↪annotation type <class 'float'>
```

CHAPTER 8

Table of Contents

Python Module Index

e

Index

R

W