Deal Documentation

@orsinium

Nov 26, 2019

Classic contracts

1 Available decorators 3

2 Installation 5

3 Quick Start 7

Deal – python library for design by contract (DbC) programming.

That's nice `assert` statements in decorators style to validate function input, output, available operations and object state. Goal is make testing much easier and detect errors in your code that occasionally was missed in tests.

- Automatic property-based tests.
- Static analysis.
- Generators and async coroutines support.
- External validators support.
- Specify allowed exceptions for function
- Invariant for all actions with class instances.
- Decorators to control available resources: forbid output, network operations, raising exceptions.
- You can disable contracts on production.

CHAPTER 1

Available decorators

CLassic DbC:

- ``@deal.pre`` <https://deal.readthedocs.io/decorators/pre.html>`_ – validate function arguments (pre-condition)
- ``@deal.post`` <https://deal.readthedocs.io/decorators/post.html>`_ – validate function return value (post-condition)
- ``@deal.ensure`` <https://deal.readthedocs.io/decorators/ensure.html>`_ – post-condition that accepts not only result, but also function arguments.
- ``@deal.inv`` <https://deal.readthedocs.io/decorators/inv.html>`_ – validate object internal state (invariant).

Take more control:

- ``@deal.module_load`` <https://deal.readthedocs.io/decorators/module_load.html>`_ – check contracts at module initialization.
- ``@deal.offline`` <https://deal.readthedocs.io/decorators/offline.html>`_ – forbid network requests
- ``@deal.raises`` <https://deal.readthedocs.io/decorators/raises.html>`_ – allow only list of exceptions
- ``@deal.reason`` <https://deal.readthedocs.io/decorators/reason.html>`_ – check function arguments that caused a given exception.
- ``@deal.silent`` <https://deal.readthedocs.io/decorators/silent.html>`_ – forbid output into stderr/stdout.

Helpers:

- ``@deal.chain`` <https://deal.readthedocs.io/decorators/chain.html>`_ – chain a few contracts in one.
- ``@deal.pure`` <https://deal.readthedocs.io/decorators/pure.html>`_ – alias for `safe`, `silent`, and `offline`.
- ``@deal.safe`` <https://deal.readthedocs.io/decorators/safe.html>`_ – forbid exceptions.

CHAPTER 2

Installation

```
python3 -m pip install --user deal
```

CHAPTER 3

Quick Start

```python
import re

import attr
import deal

REX_LOGIN = re.compile(r'^[a-zA-Z][a-zA-Z0-9]+$')

class PostAlreadyLiked(Exception):
    pass

@deal.inv(lambda post: post.visits >= 0)
class Post:
    visits: int = attr.ib(default=0)
    likes: set = attr.ib(factory=set)

    @deal.pre(lambda user: REX_LOGIN.match(user), message='invalid username format')
    @deal.raises(PostAlreadyLiked)
    @deal.chain(deal.offline, deal.silent)
    def like(self, user: str) -> None:
        if user in self.likes:
            raise PostAlreadyLiked
        self.likes.add(user)

    @deal.post(lambda result: 'visits' in result)
    @deal.post(lambda result: 'likes' in result)
    @deal.post(lambda result: result['likes'] > 0)
    @deal.pure
    def get_state(self):
        return dict(visits=self.visits, likes=len(self.likes))
```

Now, Deal controls conditions and states of the object at runtime:

1. `@deal.inv` controls that visits count in post always non-negative.
2. `@deal.pre` checks user name format. We assume that it should be validated somewhere before by some nice

forms with user-friendly error messages. So, if we have invalid login passed here, it's definitely developer's mistake.

3. `@deal.raises` says that only possible exception that can be raised is `PostAlreadyLiked`.
4. `@deal.chain(deal.offline, deal.silent)` controls that function has no network requests and has no output in stderr or stdout. So, if we are making unexpected network requests somewhere inside, deal let us know about it.
5. `deal.post` checks result format for `get_state`. So, all external code can be sure that fields `likes` and `visits` always represented in the result and likes always positive.

If code violates some condition, sub-exception of `deal.ContractError` will be raised:

```
p = Post()
p.visits = -1
# InvContractError:
```

Dive deeper on deal.readthedocs.io.

3.1 pre

Precondition – condition that must be true before function is executed. Raises `PreContractError` otherwise.

```
@deal.pre(lambda *args: all(arg > 0 for arg in args))
def sum_positive(*args):
    return sum(args)

sum_positive(1, 2, 3, 4)
# 10

sum_positive(1, 2, -3, 4)
# PreContractError:
```

It works the same for generators and async functions.

3.2 post

Postcondition – condition that must be true after function executed. Raises `PostContractError` otherwise.

```
@deal.post(lambda x: x > 0)
def always_positive_sum(*args):
    return sum(args)

always_positive_sum(2, -3, 4)
# 3

always_positive_sum(2, -3, -4)
# PostContractError:
```

For async functions it works the same. For generators validation runs for every yielded value:

```
@deal.post(lambda result: result == 2 or result % 2 == 1)
@deal.post(lambda result: result == 3 or result % 3 != 0)
```

(continues on next page)

(continued from previous page)

```
def get_primary_numbers():
    yield from (2, 3, 5, 7, 11, 13)
```

3.2.1 Motivation

Post-condition allows to make additional constraints about function result. Use type annotations to limit types of result and post-conditions to limit possible values inside given types. Let's see a few examples.

If function `count` returns count of elements that equal to given element, result is always non-negative.

```
@deal.post(lambda result: result >= 0)
def count(items: List[str], item: str) -> int:
    ...
```

Or you can make promise that your response always contains some specific fields:

```
@deal.post(lambda result: 'code' in result)
@deal.post(lambda result: 'records' in result)
def get_usernames(role):
    if role != 'admin':
        return dict(code=403, records=[])
    return dict(code=200, records=['oleg', 'greg', 'admin'])
```

3.3 ensure

Ensure is a postcondition that accepts not only result, but also function arguments. Must be true after function executed. Raises `PostContractError` otherwise.

```
@deal.ensure(lambda x, result: x != result)
def double(x):
    return x * 2

double(2)
# 4

double(0)
# PostContractError:
```

For async functions it works the same. For generators validation runs for every yielded value:

```
@deal.ensure(lambda start, end, result: start <= result < end)
def range(start, end):
    step = start
    while step < end:
        yield step
        step += 1
```

3.3.1 Motivation

Ensure allows you to simplify testing, easier check hypothesis, tell more about the function behavior. It works perfect for P vs NP like problems. In other words, for complex task when checking result correctness (even partial checking only for some cases) is much easier then calculation itself. For example:

```
from typing import List

# element at this position matches item
@deal.ensure(
    lambda items, item, result: items[result] == item,
    message='invalid match',
)
# element at this position is the first match
@deal.ensure(
    lambda items, item, result: not any(el == item for el in items[:result]),
    message='not the first match',
)
def index_of(items: List[int], item: int) -> int:
    for index, element in enumerate(items):
        if element == item:
            return index
    raise LookupError
```

Also, it's ok if you can check only some simple cases. For example, function `map` applies given function to the list. Let's check that count of returned elements is the same as the count of given elements:

```
from typing import Callable, List

@deal.ensure(lambda: items, func, result: len(result) == len(items))
def map(items: List[str], func: Callable[[str], str]) -> List[str]:
    ...
```

Or if function `choice` returns random element from the list, we can't from one run check result randomness, but can't ensure that result is an element from the list:

```
@deal.ensure(lambda items, result: result in items)
def choice(items: List[str]) -> str:
    ...
```

3.4 inv

Invariant – condition that can be relied upon to be true during execution of a program.

Invariant check condition in the next cases:

1. Before class method execution.
2. After class method execution.
3. After some class attribute setting.

```
@deal.inv(lambda post: post.likes >= 0)
class Post:
    likes = 0

post = Post()

post.likes = 10

post.likes = -10
# InvContractError:
```

(continues on next page)

(continued from previous page)

```
type(post)
# deal.core.PostInvarianted
```

3.4.1 Motivation

Make assertions about object internal state and be sure that it always true.

3.5 Exceptions

Every contract type has it's own exception type. Every exception inherited from `ContractError`. `ContractError` inherited from built-in `AssertionError`.

Custom error message for any contract can be specified by `message` argument:

```
@deal.pre(lambda name: name.lower() != 'oleg', message='user name cannot be Oleg')
def hello(name):
    print(f'hello, {name}')

hello('Oleg')
# PreContractError: user name cannot be Oleg
```

Custom exception for any contract can be specified by `exception` argument:

```
@deal.pre(lambda role: role in ('user', 'admin'), exception=LookupError)
def change_role(role):
    print(f'now you are {role}!')

change_role('superuser')
# LookupError:
```

Also, contract can return string, and this string will be used as error message:

```
def contract(name):
    if name.lower() == 'oleg':
        return 'not today, Oleg'
    if name in ('admin', 'moderator'):
        return 'this name is reservered'
    return True

@deal.pre(contract)
def register(name):
    print(f'welcome on board, {name}!')

register('Greg')
# welcome on board, Greg!

register('Oleg')
# PreContractError: not today, Oleg
```

3.6 Disable contracts on production

If you want disable contracts on production, pass `debug=True` to decorator:

```
@deal.post(lambda x: x > 0, debug=True)
def my_sum(*args):
    return sum(args)
```

If you run Python with `-O` option, contracts will be disabled. This is uses Python's `__debug__` option:

> The built-in variable `__debug__` is True under normal circumstances, False when optimization is requested (command line option -O).
>
> - Official documentation

Also, you can explicitly enable or disable contracts:

```
# disable contracts without `debug=True`
deal.switch(main=False)

# enable contracts with `debug=True`
deal.switch(debug=True)

# disable contracts with `debug=True`
deal.switch(debug=False)

# disable all contracts
deal.switch(main=False, debug=False)

# return default behavior
# (`main` enabled, `debug` the same as `__debug__`)
deal.reset()
```

3.7 Contracts chaining

You can chain any contracts:

```
@deal.pre(lambda x: x > 0)
@deal.pre(lambda x: x < 10)
def f(x):
    return x * 2

f(5)
# 10

f(-1)
# PreContractError:

f(12)
# PreContractError:
```

`@deal.post` and `@deal.ensure` contracts are resolved from bottom to top. All other contracts are resolved from top to bottom.

3.8 Validators

3.8.1 Simple contract

Regular contract can return error message instead of `False`:

```
def contract(name):
    if not isinstance(name, str):
        return 'name must be str'
    return True

@deal.pre(contract)
def f(x):
    return x * 2

f('Chris')
# 'ChrisChris'

f(4)
# PreContractError: name must be str
```

3.8.2 External validators

For a complex validation you can wrap your contract into vaa. It supports Marshmallow, WTForms, PyScheme etc. For example:

```
import deal
import marshmallow
import vaa

class Schema(marshmallow.Schema):
    name = marshmallow.fields.Str()

@deal.pre(vaa.marshmallow(Schema))
def func(name):
    return name * 2

func('Chris')
'ChrisChris'

func(123)
# PreContractError: {'name': ['Not a valid string.']}
```

3.9 Testing

Deal can automatically test your functions. First of all, your function has to be prepared:

1. All function arguments are type-annotated.
2. All exceptions that function can raise are specified in @deal.raises.
3. All pre-conditions are specified with @deal.pre.

Then use `deal.cases` to get test cases for the function. Every case is a callable object that gets no arguments. Calling it will call the original function with suppressing allowed exceptions.

3.9.1 Example

```
@deal.raises(ZeroDivisionError)
@deal.pre(lambda a, b: a > 0 and b > 0)
def div(a: int, b: int) -> float:
    return a / b


for case in deal.cases(div):
    case()
```

3.9.2 PyTest

A simple snippet to use `deal.cases` with pytest (type annotations are optional):

```
@pytest.mark.parametrize('case', deal.cases(div))
def test_div(case: deal.TestCase) -> None:
    case()
```

3.9.3 How it works

1. Deal generates random values for all function arguments with hypothesis.
2. For every arguments combination `deal.cases` returns `deal.TestCase` object.
3. `deal.TestCase` on calling is doing the next steps:
    1. Calling the original function with all decorators and contracts.
    2. Suppressing exceptions from `deal.pre` and `deal.raises`. In that case `typing.NoReturn` returned.
    3. Validating type of the function result if it's annotated.
    4. Returning the function result.

All uncatched exceptions are raised: everything you forgot to specify in `@deal.raises`, `deal.PostContractError`, `deal.OfflineContractError` etc.

3.9.4 Configuring

Specify samples count (50 by default):

```
deal.cases(div, count=20)
```

Explicitly specify arguments to pass into the function:

```
deal.cases(div, kwargs=dict(b=3))
```

3.9.5 deal.TestCase

`deal.TestCase` object has the next attributes:

- `case.args` – tuple of positional arguments that will be passed into the original function.

- `case.kwargs` – dict of keyword arguments that will be passed into the original function.
- `case.func` – the original function itself
- `case.exceptions` – tuple of all exceptions that will be ignored.

3.9.6 Practical example

The best case for Contract-Driven Development is when you have a clear business requirements for part of code. Write these requirements as contracts, and then write a code that satisfy these requirements.

In this example, we will implement `index_of` function that returns index of the given element in the given list. Let's think about requirements:

1. Function accepts list of elements (let's talk about list of integers), one element, and returns index.
2. Result is in range from zero to the length of the list.
3. Element by given index (result) is equal to the given element.
4. If there are more than one matching element in the list, we'll return the first one.
5. If there is no matching elements, we'll raise `LookupError`.

And now, let's convert it from words into the code:

```
from typing import List, NoReturn
import deal

# if you have more than 2-3 contracts,
# consider moving them from decorators into separate variable
# like this:
contract_for_index_of = deal.chain(
    # result is an index of items
    deal.post(lambda result: result >= 0),
    deal.ensure(lambda items, item, result: result < len(items)),
    # element at this position matches item
    deal.ensure(
        lambda items, item, result: items[result] == item,
        message='invalid match',
    ),
    # element at this position is the first match
    deal.ensure(
        lambda items, item, result: not any(el == item for el in items[:result]),
        message='not the first match',
    ),
    # LookupError will be raised if no elements found
    deal.raises(LookupError),
    deal.reason(LookupError, lambda items, item: item not in items)
)
```

Now, we can write a code that satisfies our requirements:

```
@contract_for_index_of
def index_of(items: List[int], item: int) -> int:
    for index, el in enumerate(items):
        if el == item:
            return index
    raise LookupError
```

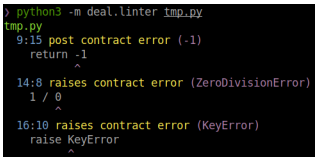
And tests, after all, the easiest part. Let's make it a little bit interesting and in the process show all valid samples:

```
# test and make examples
for case in deal.cases(index_of, count=1000):
    # run test case
    result = case()
    if result is not NoReturn:
        # if case is valid show it
        print(f"index of {case.kwargs['item']} in {case.kwargs['items']} is {result}")
```

3.10 Static analysis

Deal can do static checks for functions with contracts to catch trivial mistakes. Use flake8 or flakehell to run it.

Another option is to use built-in CLI from deal: `python3 -m deal.linter`. I has beautiful colored output by default. Use `--json` option to get compact JSON output. Pipe output into jq to beautify JSON.



3.10.1 Codes

Code	Message
DEAL001	do not use `from deal import ...`, use `import deal` instead
DEAL011	post contract error
DEAL012	raises contract error
DEAL013	silent contract error

3.11 module_load

This function allows you to control what can happen at module load stage.

Usage:

1. Call `deal.activate()` before importing anything.
2. Call `deal.module_load()` in any place at module level in all modules that should be tested. Pass inside all contracts that should be controlled. By design, only contracts from deal without arguments are supported.

3.11.1 Example

`__init__.py`:

```
import deal

deal.activate()

from .other import something
```

other.py:

```
import deal
import something_else

deal.module_load(deal.silent, deal.safe, deal.offline)

something = 1
print(1)  # contract violation! deal.SilentContractError will be raised
```

3.11.2 How it works

1. Calling `deal.activate` registers import finder and loader. From now, all imported files will be checked by deal.
2. The loader reads imported file, generates AST for it, and looks for `deal.module_load` calling.
3. If loader found `deal.module_load` in the module, it extracts contracts from it.
4. If all contracts are valid (imported from deal and have no arguments), loader loads the module with contracts activated.

3.11.3 Motivation

This contract is inspired by article Python at Scale: Strict Modules. A module loading should be fast, pure, and safe. This function allows to enforce it.

3.12 offline

Offline function cannot do network requests. This is achieved by patching `socket.socket`. So, don't use it into asynchronous or multi-threaded code.

```
@deal.offline
def ping(host):
    if host == 'localhost':
        return True
    response = requests.head('http://' + host)
    return response.ok

ping('localhost')
# True

ping('ya.ru')
# OfflineContractError:
```

It works the same for generators. For async functions keep in mind patching.

3.12.1 Motivation

Sometimes, your code are doing unexpected network requests. Use `@offline` to catch these cases to do code optimization if possible.

Bad:

```
def get_genres():
    response = requests.get('http://example.com/genres.txt')
    response.raise_for_status()
    return response.text.splitlines()

def valid_genre(genre):
    ...
    return genre in get_genres()

def validate_genres(genres):
    return all(valid_genre(genre) for genre in genres)
```

Good:

```
def get_genres():
    response = requests.get('http://example.com/genres.txt')
    response.raise_for_status()
    return response.text.splitlines()

@deal.offline
def valid_genre(genre, genres):
    ...
    return genre in genres

def validate_genres(genres):
    existing_genres = get_genres()
    return all(valid_genre(genre, existing_genres) for genre in genres)
```

3.13 raises

Specifies which exceptions function can raise.

```
@deal.raises(ZeroDivisionError)
def divide(*args):
    return sum(args[:-1]) / args[-1]

divide(1, 2, 3, 6)
# 1.0

divide(1, 2, 3, 0)
# ZeroDivisionError: division by zero

divide()
# IndexError: tuple index out of range
# The above exception was the direct cause of the following exception:
# RaisesContractError:
```

It works the same for generators and async functions.

3.13.1 Motivation

Exceptions are the most explicit part of Python. Any code can raise any exception. None of the tools can say you which exceptions can be raised in some function. However, sometimes you can infer it yourself and say it to other people. And `@deal.raises` will remain you if function has raised something that you forgot to specify.

Also, it's the most important decorator for autotesting. Deal won't fail tests for exceptions that was marked as allowed with `@deal.raises`.

Bad:

```
ROLES = {
    'admin': 'admin',
    'oleg': 'user',
    'greg': 'user',
}

def get_role(username):
    return ROLES[username]
```

Good:

```
ROLES = {
    'admin': 'admin',
    'oleg': 'user',
    'greg': 'user',
}

@deal.raises(KeyError)
def get_role(username):
    return ROLES[username]
```

3.14 reason

Checks condition if exception was raised.

```
@deal.reason(ZeroDivisionError, lambda a, b: b == 0)
def divide(a, b):
    return a / b
```

It works the same for generators and async functions.

3.14.1 Motivation

This is the @deal.ensure for exceptions. It works perfect when it's easy to check correctness of conditions when exception is raised.

For example, if function `index_of` returns index of the first element that equal to the given element and raises `LookupError` if element is not found, we can check that if `LookupError` is raised, element not in the list:

```
@deal.reason(LookupError, lambda items, item: item not in items)
def index_of(items: List[int], item: int) -> int:
    for index, el in enumerate(items):
        if el == item:
            return index
    raise LookupError
```

3.15 silent

Silent function cannot write anything into stdout. This is achieved by patching `sys.stdout`. So, don't use it into asynchronous or multi-threaded code.

```
@deal.silent
def has_access(role):
    if role not in ('user', 'admin'):
        print('role not found')
        return False
    return role == 'admin'

has_access('admin')
# True

has_access('superuser')
# SilentContractError:
```

It works the same for generators. For async functions keep in mind patching.

3.15.1 Motivation

If possible, avoid any output from function. Direct output makes debugging and re-usage much more difficult. Of course, there are some exceptions:

1. Entry-points that communicate with user and never should be used from other code.
2. Logging. Logging also output, but this output makes our life easier. However, be sure, you're not using logging for something important that should be checked from other code or tested.

Bad:

```
def say_hello(name):
    print('Hello, {name}')

def main():
    say_hello(sys.argv[1])
```

Good:

```
@deal.silent
def make_hello(name):
    return 'Hello, {name}'

def main(argv=None):
    if argv is None:
        argv = sys.argv[1:]
    print(make_hello(argv[0]))
```

3.16 chain

Beautiful way to apply a few short decorators to a function.

```
@deal.chain(deal.safe, deal.silent)
def show_division(a, b):
    print(a / b)

show_division(1, 2)
# SilentContractError:

show_division(1, 0)
# RaisesContractError:
```

3.16.1 Motivation

Get rid of long chains of decorators because it looks like centipede. Use `@chain` to place short decorators horizontally.

3.17 pure

Pure function cannot do network requests, write anything into stdout or raise any exceptions. It gets some parameters and returns some result. That's all. This is alias for `chain(safe, silent, offline)`.

```
@deal.pure
def show_division(a, b):
    print(a / b)

show_division(1, 2)
# SilentContractError:

show_division(1, 0)
# RaisesContractError:
```

3.17.1 Motivation

Explicitly mark pure functions in your code. Pure functions easier to test and safer to use.

3.18 safe

Safe function cannot raise any exceptions. Alias for raises without arguments.

```
@deal.safe
def divide(a, b):
    return a / b

divide(1, 2)
# 0.5

divide(1, 0)
# ZeroDivisionError: division by zero
# The above exception was the direct cause of the following exception:
# RaisesContractError:
```

3.18.1 Motivation

Can you be sure that some function never raises exceptions? Make promise about this, and `@deal.safe` will help you to control it.