Deal Documentation

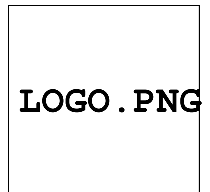
@orsinium

Nov 29, 2021

GETTING STARTED

1	1
Python Module Index	59
Index	61

CHAPTER ONE



A Python library for design by contract (DbC) and checking values, exceptions, and side-effects. In a nutshell, deal empowers you to write bug-free code. By adding a few decorators to your code, you get for free tests, static analysis, formal verification, and much more. Read intro to get started.

1.1 Features

- Classic DbC: precondition, postcondition, invariant.
- Tracking exceptions and side-effects.
- Property-based testing.
- Static checker.
- Integration with pytest, flake8, sphinx, and hypothesis.
- Type annotations support.
- External validators support.
- Contracts for importing modules.
- Can be enabled or disabled on production.
- Colorless: annotate only what you want. Hence, easy integration into an existing project.
- Colorful: syntax highlighting for every piece of code in every command.
- Memory leaks detection: deal makes sure that pure functions don't leave unexpected objects in the memory.
- DRY: test discovery, error messages generation.
- Partial execution: linter executes contracts to statically check possible values.
- Formal verification: prove that your code works for all input (or find out when it doesn't).
- Fast: each code change is benchmarked and profiled.
- Reliable: the library has 100% test coverage, partially verified, and runs on production by multiple companies since 2018.

1.2 Deal in 30 seconds

```
# the result is always non-negative
@deal.post(lambda result: result >= 0)
# the function has no side-effects
@deal.pure
def count(items: List[str], item: str) -> int:
    return items.count(item)

# generate test function
test_count = deal.cases(count)
```

Now we can:

- Run `python3 -m deal lint` or `flake8` to statically check errors.
- Run `python3 -m deal test` or `pytest` to generate and run tests.
- Just use the function in the project and check errors in runtime.

Read more in the documentation.

1.3 Installation

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

1.4 Contributing

Contributions are welcome! A few ideas what you can contribute:

- Add new checks for the linter.
- Improve documentation.
- Add more tests.
- Improve performance.
- Found a bug? Fix it!
- Made an article about deal? Great! Let's add it into the `README.md`.
- Don't have time to code? No worries! Just tell your friends and subscribers about the project. More users -> more contributors -> more cool features.

Thank you :heart:

1.4.1 Intro

About contracts

Deal is a powerful library for writing and testing contracts.

1. **Testing** is for checking exact values. You assume that for some exact input values and exact state the function returns an exact known value. For example, `sum(2, 3) == 5`.
2. **Typing** is for checking sets of values. You state that the function accepts only some class of values and returns a class of values. For example, `sum(float, float) -> float`.
3. **Property-based testing** is for checking conditions for a set of values. It's like typing but it actually checks not classes of values but exact values from the class. For example, if property is "sum of 2 positive numbers is also positive", property-based tests will take random positive numbers, call the function and check that result is also positive.
4. **Contracts** are a powerful mix of typing and property-based testing.
   1. Like type annotations, contracts are part of the function signature, and can be checked statically.
   2. Like properties, contacts allow you to specify any conditions, and the framework will take care of choosing exact values and checking call results.

So, think about it as typing on steroids. However, Deal doesn't try to replace type annotations (mypy isn't perfect but it's hard to do better) but instead empowers them, says more about possible values and their properties.

Read *Contract-Driven Development* section if you want to know more why contracts are cool.

Open-world assumption

Deal can tell you if something goes wrong but can't tell you if something can't go wrong. It is known as open-world assumption. For example, if the function explicitly raises an exception or does it almost on every input, Deal will tell you about it. However, if the function does it somewhere deep inside of call stack and only on one value from million, chances that it will be caught are small. So, if you say "this function can raise ValueError" but Deal doesn't see it anywhere, it will trust you and don't argue about it. Deal assumes that the developer is smart and can see something that the framework can't.

Writing contracts

The next 3 parts of the documentation tell how to check different kinds of things that can happen when you call a function:

1. *Values* – arguments of the function and return values. That's all what pure functional languages have but Python is different.
2. *Exceptions* – be aware of where your code execution can stop.
3. *Side-effects* – when function mutates global values, does request to database or remote server, or even imports a module.

Checking contracts

There are a multiple ways to validate contracts:

1. *Runtime*. Call the functions, do usual tests, just play around with the application, deploy it to staging, and Deal will check contracts in runtime. Of course, you can disable contracts on production.
2. *Tests*. Deal is easily integrates with PyTest or any other testing framework. It does property-based testing for functions with contracts. Also, deal has `test` CLI command to find and run all pure functions in the project.
3. *Linter*. The most amazing part of Deal. It statically checks constant values in the code, does values inference, contracts partial execution, propagates exceptions and side-effects. Deal has `lint` CLI command for it and flake8 integration.
4. **Experimental:** *Formal verification*. The most powerful but limited idea in the whole project. Deal can turn your code into mathematical expressions and verify its correctness.
5. **Experimental:** *CrossHair*. Third-party verifier-driven fuzzer, something between deal's testing and verification.

Dive deeper

It's not "advanced usage", there is nothing advanced or difficult. It's about writing better contracts or saving a bit of time. Not important but very useful. So, don't be afraid to dive into this section!

1. *More on writing contracts* gives you additional tools to reuse and simplify contracts.
2. *Contracts for modules* allow you to control what happens at the module load (import) time.
3. *Dispatch* is a way to combine multiple implementations for a function into one based on pre-conditions.
4. *Documentation* provides information on generating documentation for functions with contracts (using Sphinx).
5. *Stubs* is a way to store some contracts in a JSON file instead of the source code. It can be helpful for third-party libraries. Some stubs already inside Deal.
6. *More on testing* provides information on finding memory leaks and tweaking tests generation.
7. *Recipes* is the place to learn more about best practices of using contracts.

1.4.2 Contract-Driven Development

Let's take for example an incredibly simple code and imagine that it's incredibly complicated logic.

```
def cat(left, right):
    """Concatenate two given strings.
    """
    return left + right
```

Tests

How can we be sure this code works? No, it's not obvious. Remember the rules of the game, we have an incredibly complicated realization. So, we can't say it works or not while we haven't tested it.

```
def test_cat():
    result = cat(left='abc', right='def')
    assert result == 'abcdef'
```

Now, run pytest:

```
pytest cat.py
```

It passes. So, our code works. Right?

Table tests

Wait, but what about corner cases? What if one string is empty? What if both strings are empty? What if we have only one character in both strings? We need check more values and this is where table driven tests will save our time. In pytest, we can use @pytest.mark.parametrize to make such tables.

```
import pytest

@pytest.mark.parametrize('left, right, expected', [
    ('a', 'b', 'ab'),
    ('', '', ''),
    ('', 'b', 'b'),
    ('a', '', 'a'),
    ('text', 'check', 'textcheck'),
])
def test_cat(left, right, expected):
    result = cat(left=left, right=right)
    assert result == expected
```

Properties

Table tests can be enormously long, and for every test case, we have to manually calculate the expected result. For complicated code, it's a lot of work. Can we do it better and think and write less? Yes, we can instead of *expected result* talk about *expected properties of the result*. The big difference is the result is different for different input values, but properties always the same. The coolest thing is in most cases you already know result properties, it is the business requirements, and your code is no more than the implementation of these requirements.

So, what are the properties of our function?

1. The result string starts with the first given string.
2. The result string ends with the second given string.
3. Result string has the length equal to the sum of lengths of given strings.

Now, we can check these properties for the result instead of checking particular values.

```
@pytest.mark.parametrize('left, right', [
    ('a', 'b'),
    ('', ''),
    ('', 'b'),
    ('a', ''),
    ('text', 'check'),
])
def test_cat(left, right):
    result = cat(left=left, right=right)
    assert result.startswith(left)
    assert result.endswith(right)
    assert len(result) == len(left) + len(right)
```

Hypothesis

We've tested a few corner cases but not all of them. What about Unicode strings? What if one string is Unicode, but another one isn't? What about spaces? What if we have a string termination symbol somewhere? What if both strings contain only digits (the place where JS always surprises)? It's so hard to find examples for all possible cases where something can go wrong. In theory, you even can't say that it works while you haven't checked **all** possible values (that impossible even for our simple function). So, instead of trying to figure out all possible nightly values we can ask the machine to do so. This is where the property-based testing comes in. In Python, we have a great tool hypothesis that can generate test examples for us:

```
import hypothesis
from hypothesis import strategies

@hypothesis.given(left=strategies.text(), right=strategies.text())
def test_cat(left, right):
    result = cat(left=left, right=right)
    assert result.startswith(left)
    assert result.endswith(right)
    assert len(result) == len(left) + len(right)
```

Type annotations

Another one cool thing in Python we have to talk about before moving further is type annotations:

```
def cat(left: str, right: str) -> str:
    return left + right
```

Type annotations aren't perfect and can be too complicated. However, what is most important is now humans and machines know much more about your code. You can run mypy and check that you haven't made type errors. And the thing is it's not only about catching type errors. Now we can use hypothesis-auto wrapper around hypothesis. It will infer parameters types and explain names and types of parameters to Hypothesis. So, instead of writing `hypothesis.given(left=strategies.text(), right=strategies.text())` we can just say `hypothesis_auto.auto_pytest(cat)`.

```
import hypothesis_auto

@hypothesis_auto.auto_pytest(cat)
def test_cat(test_case):
    result = test_case()
    left = test_case.parameters.kwargs['left']
    right = test_case.parameters.kwargs['right']
    assert result.startswith(left)
    assert result.endswith(right)
    assert len(result) == len(left) + len(right)
```

It looks longer because now parameters are placed inside the long name `test_case.parameters.kwargs` but the most important thing here is we don't talk about function inputs at all, the machine does everything. The test isn't about any values of the function anymore but only about the function properties.

Contracts

Can we make it even simpler? Not really. The implementation can produce some values, and the machine can infer some properties of the result. However, someone else must say which properties are good and expected, and which are not. However, there is something else about our properties that we can do better. At this stage we have type annotations and, to be honest, they are just kind of properties. Annotations say "the result is a text", and our test properties clarify the length of the result, it's prefix and suffix. However, the difference is type annotations are the part of the function itself. It gives some benefits:

1. The machine can check statically, without the actual running of the code.
2. The human can see types (think "possible values set") for arguments and the result.

And Deal can make the same for function properties:

```
import deal

@deal.ensure(lambda left, right, result: result.startswith(left))
@deal.ensure(lambda left, right, result: result.endswith(right))
@deal.ensure(lambda left, right, result: len(result) == len(left) + len(right))
def cat(left: str, right: str) -> str:
    return left + right
```

Or using short signatures:

```
import deal

@deal.ensure(lambda _: _.result.startswith(_.left))
@deal.ensure(lambda _: _.result.endswith(_.right))
@deal.ensure(lambda _: len(_.result) == len(_.left) + len(_.right))
def cat(left: str, right: str) -> str:
    return left + right
```

Now, it's not just properties, but contracts. They can be checked in the runtime, simplify tests, tell humans about the function behavior. And tests for this implementation are trivial:

```
test_cat = deal.cases(cat)
```

Contracts for machines

The most exciting thing is deal can check contracts statically, like mypy checks annotations. However, contracts can be any code while types are standardized and limited. Although the machine can't check everything (yet), it can catch some trivial cases. For example:

```
@deal.post(lambda result: 0 <= result <= 1)
def sin(x):
    return 2
```

And when we run *deal linter* on this code, we see contract violation error:

```
 flake8 --show-source sin.py
sin.py:6:5: DEAL011: post contract error
    return 2
    ^
```

1.4.3 References

This page provides a quick navigation by the documentation in case if you're looking for something specific.

Decorators

decorator	reference	documentation
`@deal.chain`	*deal.chain*	*More on writing contracts* / *deal.chain*
`@deal.dispatch`	*deal.dispatch*	*Dispatch*
`@deal.ensure`	*deal.ensure*	*Values* / *deal.ensure*
`@deal.example`	*deal.example*	*Documentation* / *deal.example*
`@deal.has`	*deal.has*	*Side-effects*
`@deal.inherit`	*deal.inherit*	*More on writing contracts* / *deal.inherit*
`@deal.inv`	*deal.inv*	*Values* / *deal.inv*
`@deal.post`	*deal.post*	*Values* / *deal.post*
`@deal.pre`	*deal.pre*	*Values* / *deal.pre*
`@deal.pure`	*deal.pure*	–
`@deal.raises`	*deal.raises*	*Exceptions* / *deal.raises*
`@deal.reason`	*deal.reason*	*Exceptions* / *deal.reason*
`@deal.safe`	*deal.safe*	–

Functions

decorator	reference	documentation
`deal.activate`	*deal.activate*	*Contracts for modules*
`deal.autodoc`	*deal.autodoc*	*Documentation* / *Sphinx autodoc*
`deal.cases`	*deal.cases*	*Tests*
`deal.catch`	*deal.catch*	*Documentation* / *deal.example*
`deal.disable`	*deal.disable*	*Runtime* / *Contracts on production*
`deal.enable`	*deal.enable*	*Runtime* / *Contracts on production*
`deal.implies`	*deal.implies*	–
`deal.module_load`	*deal.module_load*	*Contracts for modules*
`deal.reset`	*deal.reset*	*Runtime* / *Contracts on production*

Exceptions

decorator	reference	documentation
`deal.ContractError`	*deal.ContractError*	*Values* / *Exceptions*
`deal.ExampleContractError`	*deal.ExampleContractError*	–
`deal.InvContractError`	*deal.InvContractError*	*Values* / *Exceptions*
`deal.MarkerError`	*deal.MarkerError*	–
`deal.NoMatchError`	*deal.NoMatchError*	*Dispatch*
`deal.OfflineContractError`	*deal.OfflineContractError*	–
`deal.PostContractError`	*deal.PostContractError*	*Values* / *Exceptions*
`deal.PreContractError`	*deal.PreContractError*	*Values* / *Exceptions*
`deal.RaisesContractError`	*deal.RaisesContractError*	–
`deal.ReasonContractError`	*deal.ReasonContractError*	–
`deal.SilentContractError`	*deal.SilentContractError*	–

CLI commands

command	reference	documentation
`decorate`	*decorate*	*More on writing contracts* / *Generating contracts*
`lint`	*lint*	*Linter* / *Built-in CLI command*
`memtest`	*memtest*	*More on testing* / *Finding memory leaks*
`prove`	*prove*	*Formal verification*
`stub`	*stub*	*Stubs*
`test`	*test*	*Tests* / *CLI*

Integrations

tool	github	integration docs
atheris	google/atheris	*More on testing* / *Fuzzing*
flake8	PyCQA/flake8	*Linter* / *flake8*
hypothesis	HypothesisWorks/hypothesis	*More on testing* / *Custom strategies*
mypy	python/mypy	*More on writing contracts* / *Typing*
pytest	pytest-dev/pytest	*Tests*
sphinx	sphinx-doc/sphinx	*Documentation* / *Sphinx autodoc*

Articles

- Make tests a part of your app

Projects integrating deal

- CrossHair
- flake8-functions-names

1.4.4 Values

deal.pre

Precondition – condition that must be true before function is executed.

```
@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: expected all(arg > 0 for arg in args) (where args=(1, 2, -3, 4))
```

deal.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:
```

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.

deal.ensure

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

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

double(2)
# 4

double(0)
# PostContractError: expected x != result (where result=0, x=0)
```

Ensure is the shining star of property-based testing. 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 the calculation itself.

deal.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: expected post.likes >= 0
```

(continues on next page)

(continued from previous page)

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

assert

Good old assert statement is also kind of a contract. It is good for checking intermediate state inside a function. Also, it is similar to other contracts since deal mimics `assert` behavior: all contracts are *disabled on production* and raise AssertionError in case of the contract violation. Also, *deal linter* checks `assert` statements to be True.

```
def do_something(a):
    result = something_else(a)
    assert result > 0
    return another_thing(result)
```

Exceptions

Every contract type raises it's own exception type, inherited from `ContractError` (which is inherited from built-in `AssertionError`):

contract	exception
pre	PreContractError
post	PostContractError
ensure	PostContractError
inv	InvContractError

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:
```

However, thumb-up rule is to avoid catching exceptions from contracts. Contracts aren't part of business logic but it's validation. Hence contract error means business logic violation and execution should be stopped to avoid doing something not predicted and even dangerous.

Chaining contracts

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
```

(continues on next page)

(continued from previous page)

```
f(-1)
# PreContractError: expected x > 0 (where x=-1)

f(12)
# PreContractError: expected x < 10 (where x=12)
```

`@deal.post` and `@deal.ensure` contracts are resolved from bottom to top. All other contracts are resolved from top to bottom. This is because of how wrapping works: before calling function we go down by contracts list, after calling the function we go back, up by call stack.

Generators and async functions

Contracts mostly support generators (`yield`) ans async functions:

contract	yield	async
pre	yes	yes
post	yes (checks every yielded value)	yes
ensure	yes (checks every yielded value)	yes
inv	partially (before execution)	partially (before execution)

1.4.5 Exceptions

deal.raises

`@deal.raises` specifies which exceptions the 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:
```

`@deal.raises()` without exceptions specified means that function raises no exception.

deal.reason

Checks condition if exception was raised.

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

Motivation

Exceptions are the most implicit 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 an important decorator for autotesting. Deal won't fail tests for exceptions that were marked as allowed with `@deal.raises`.

1.4.6 Side-effects

deal.has

`@deal.has` is a way to specify markers for a function. Markers are tags about kinds of side-effects which the function has. For example:

```
@deal.has('stdout', 'database')
def say_hello(id: int) -> None:
    user = get_user(id=id)
    print(f'Hello, {user.name}')
```

You can use any markers you want, and Deal will check that if you call a function with some markers, they are specified for the calling function as well. In the example above, `print` function has marker `stdout`, so it must be specified in markers of `say_hello` as well.

Motivation

Every application has side-effects. It needs to store data, to communicate with users. However, every side-effect makes testing and debugging much harder: it should be mocked, intercepted, cleaned after every test. The best solution is to have functional core and imperative shell. So, the function above can be refactored to be pure:

```
import sys

# now this is pure
@deal.has()
def make_hello(user) -> str:
    return f'Hello, {user.name}'

# and the main function takes care of all impure things
@deal.has('stdout', 'database')
def main(stream=sys.stdout):
    ...
    user = get_user(id=id)
    hello = make_hello(user=user)
    print(make_hello, file=stream)
```

Built-in markers

Deal already know about some markers and will report if they are violated:

code	marker	allows
DEAL041	`global`	`global` and `nonlocal`
DEAL042	`import`	`import`
DEAL043	`io`	everything below
DEAL044	– `read`	read a file
DEAL045	– `write`	write into a file
DEAL046	– `stdout`	`sys.stdout` and `print`
DEAL047	– `stderr`	`sys.stderr`
DEAL048	– `network`	network communications, `socket`
DEAL049	– `stdin`	`sys.stdin`
DEAL050	– `syscall`	system calls: `subprocess`, `os`
DEAL055	`random`	functions from `random` module
DEAL056	`time`	accessing system time

Runtime

Some of the markers are checked at runtime:

- If any of `io`, `network`, or `socket` is specified, `deal.has` will allow network operations. Otherwise, it will patch socket blocking all network requests. If the function tries to use the network, `OfflineContractError` is raised.
- If any of `io`, `print`, or `stdout` is specified, `deal.has` will allow using stdout. Otherwise, it will patch sys.stdout. If the function tries to use it, `SilentContractError` is raised.
- If any of `io` or `stdout` is specified, `deal.has` will do the same as for `stdout` marker but for sys.stderr

```
@deal.has()
def f():
    print('hello')

f()
# SilentContractError:
```

Other markers aren't checked in runtime yet but only checked by the *linter*.

Markers are properties

Markers and exceptions are properties of a function and don't depend on conditions. That means, if a function only sometimes in some conditions does io operation, the function has `io` marker regardless of possibility of hitting this condition branch. For example:

```
import deal

def run_job(job_name: str, silent: bool):
    if not silent:
        print('job started')
    ...

@deal.has()  # must have 'stdout' here.
```

(continues on next page)

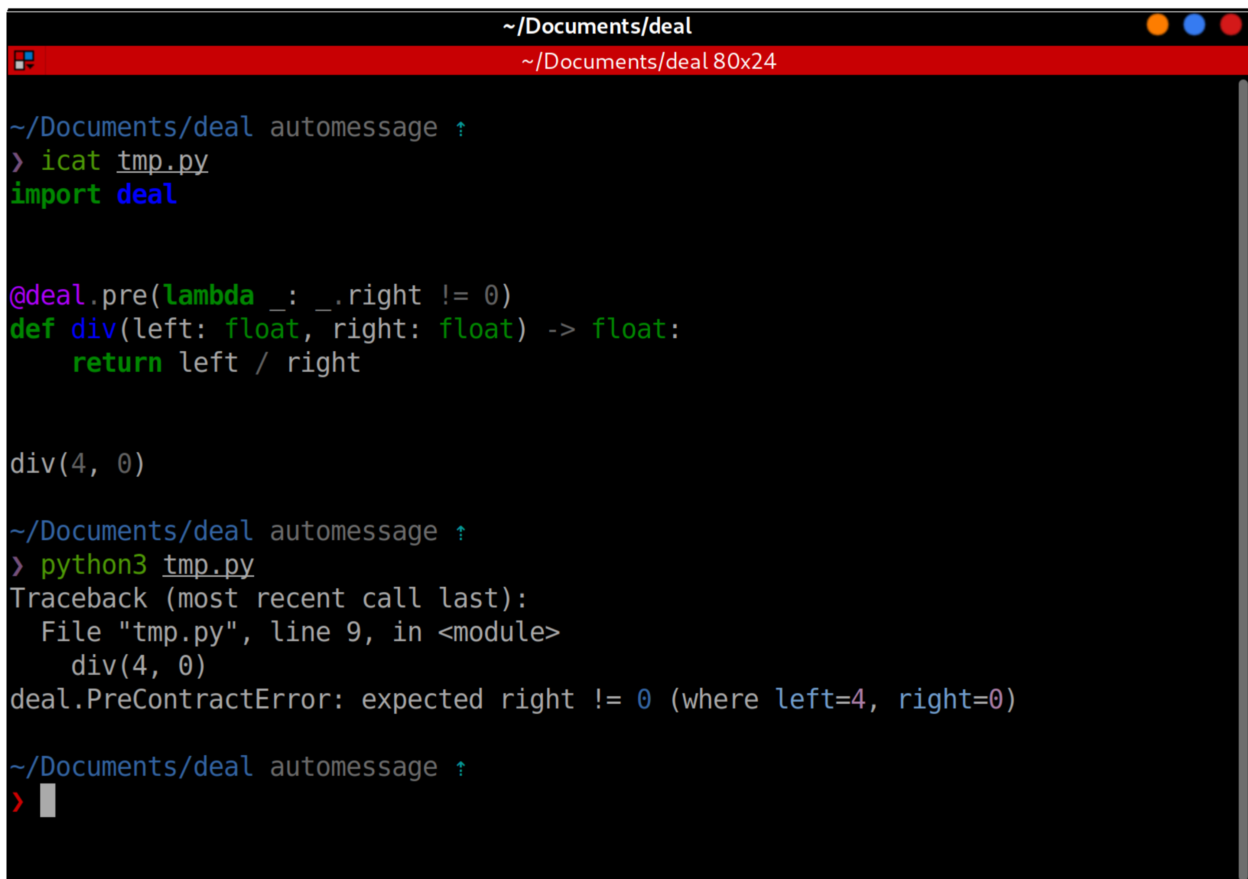
(continued from previous page)

```
def main():
    job_name = 'hello'
    run_job(job_name, silent=True)
    return 0
```

If we run *linter* on the code above, it will fail with "DEAL046 missed marker (stdout)" message. `main` function calls `run_job` with `silent=True` so `print` will not be called when calling `main`. However, `run_job` function has an implicit `stdout` marker, and `main` calls this function so it must have this marker as well.

1.4.7 Runtime

Call the functions, do usual tests, just play around with the application, deploy it to staging, and Deal will check contracts in runtime. On contract violation, deal raises an exception. In general, you shouldn't ever catch these exceptions because contracts must be never violated. Contract violation means a bug.



Contracts on production

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

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

Also, you can explicitly enable or disable contracts:

```
# disable all contracts
deal.disable()

# enable all contracts
deal.enable()

# restore the default behavior
# (enabled if `__debug__` is True, disabled otherwise)
deal.reset()
```

Colors

If no error message or custom exception specified for a contract, deal will show contract source code and passed parameters as the exception message. By default, deal highlights syntax for this source code. If your terminal doesn't support colors (which is possible on CI), you can specify `NO_COLOR` environment variable to disable syntax highlighting:

```
export NO_COLOR=1
```

See no-color.org for more details.

1.4.8 Tests

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`*.

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

Then you can use *`deal.cases`* to generate test cases for the function. This is a polimorphic object that can be used in many ways.

Here is the shortest way to create a test:

```
test_div = deal.cases(div)
```

It is enough for pytest to find and run the test. Or you can run it manually by just calling it: `test_div()`.

However, it is not scalable. What if we want to use a pytest fixture? What if we need to prepare something before running the test case? Or what if we want to check additional conditions? So, let's make a proper test function:

```
# type annotations below are optional
@deal.cases(div)
def test_div(case: deal.TestCase) -> None:
    case()
```

In this example, when we (or pytest) call `test_div()`, deal will generate test cases (using hypothesis) and run the function body for every case. The test function itself decides when to execute the test case. Here `case` is an instance of *deal.TestCase* class. This form works as expected with pytest fixtures if the test case is the first argument and has the name `case`.

CLI

There is a CLI command named `test`. It extracts `deal.pure` and `@deal.has()` (without arguments) wrapped functions and runs `deal.cases` powered tests for it.

```
python3 -m deal test project/*.py
```

The command is helpful when you don't have tests for some pure functions yet but want to get an early feedback.

For every ran function, deal calculates and shows coverage. This is a helpful indication on how good deal was at finding the correct input values.

```
~/Documents/deal
~/Documents/deal 80x24
❯ python3 -m deal test --count=50 examples
running examples/count.py
  running count
    coverage 100%
running examples/min.py
  running my_min
    coverage 100%
running examples/choice.py
  running choice
    coverage 100%
running examples/concat.py
  running concat
    coverage 100%
running examples/index_of.py
  running index_of
    coverage 100%
running examples/div.py
  running div1
    coverage 100%
  running div2
    coverage 100%

~/Documents/deal coverage*
❯
```

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))
```

See *deal.cases* API documentation and *More on testing* for details.

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),
    # no side-effects
    deal.has(),
)
```

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

```python
@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:

```python
# test and make examples
@deal.cases(index_of, count=1000)
def test_div(case):
    # run test case
    result = case()
    if result is not NoReturn:
        # if no exceptions was raised, print the result
        print(f"index of {case.kwargs['item']} in {case.kwargs['items']} is {result}")
```

1.4.9 Linter

Deal can do static checks for functions with contracts to catch trivial mistakes. Don't expect it to find much. Static analysis in dynamic language is hard but deal tries its best to help you. Add the linter on your CI and it will help you to find bugs.

flake8

Most probably, you already use flake8, so this option should suit best for you. Deal has built-in flake8 plugin which will be autimatically discovered if you install flake8 and deal in the same environment.

```
python3 -m pip install --user flake8 deal
python3 -m flake8
```

Built-in CLI command

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

Since this is ad-hoc solution, it has a bit more beautiful colored output.

```
~/Documents/deal
~/Documents/deal 80x39

~/Documents/deal automessage* ⇡
❯ icat examples/min.py
from typing import List, TypeVar

import deal
import pytest


T = TypeVar('T')


@deal.pre(lambda items: len(items) > 0)
@deal.has()
def my_min(items: List[T]) -> T:
    return min(items)


@deal.has('stdout')
def example():
    # good
    print(my_min([3, 1, 4]))
    # bad
    print(my_min([]))


@pytest.mark.parametrize('case', deal.cases(my_min))
def test_min(case):
    case()

~/Documents/deal automessage* ⇡
❯ python3 -m deal lint examples/min.py
examples/min.py
  21:10 DEAL011 pre contract error ([])
    print(my_min([]))
          ^

~/Documents/deal automessage* ⇡
❯
```

Codes

General:

Code	Message
DEAL001	do not use `from deal import ...`, use `import deal` instead
DEAL002	ensure contract must have `result` arg

Contracts:

Code	Message
DEAL011	pre contract error
DEAL012	post contract error
DEAL013	example violates contract
DEAL021	raises contract error
DEAL031	assert error

Markers:

Code	Message
DEAL041	missed marker (global)
DEAL042	missed marker (import)
DEAL043	missed marker (io)
DEAL044	missed marker (read)
DEAL045	missed marker (write)
DEAL046	missed marker (stdout)
DEAL047	missed marker (stderr)
DEAL048	missed marker (network)
DEAL049	missed marker (stdin)
DEAL050	missed marker (syscall)
DEAL055	missed marker (random)
DEAL056	missed marker (time)

Partial execution

To check `pre` and `post` contracts, linter can partially execute them. For example:

```
import deal

@deal.post(lambda r: r != 0)
def f():
    return 0
```

Try to run linter against the code above:

```
$ python3 -m deal lint tmp.py
tmp.py
  6:11 DEAL012 post contract error (0)
    return 0
```

Hence there are some rules to make your contracts linter-friendly:

- Avoid side-effects, even logging.

- Avoid external dependencies (functions and contants defined outside of the contract).
- Keep them as small as possible. If you have a few different things to check, make separate contracts.

Linter silently ignores contract if it cannot be executed.

1.4.10 Formal verification

> **Warning:** This feature is **experimental** and always will be. The API is stable and verification is reliable but it always will work only for some simple cases.

Deal has a built-in formal verifier. That means, deal turns your code into a formal theorem and then proves that it is formally correct (or finds a counter-example when it is not). Turning wild Python code into mathematical expressions is hard, so application of the verifier is limited. Still, you should try it. It will work for only 1% of your code but when it does work, it finds actual bugs.

```
python3 -m deal prove project/
```

How it works

Prerequisites for code to be verified:

- It is a function or `@staticmethod`.
- It is written on pure Python and calls only pure Python code.
- Every argument is type annotated.
- Even if everything above is satisfied, the functon still can be skipped because a feature it uses is not supported yet.

Below, we use the following terms:

- **counter-example** means a combination of input arguments that leads to theorem violation.
- **given** means that this is an axiom the theorem uses. Counter-example must satisfy to the given conditions.
- **expected** means that this is an assertion that theorem tries to break. Counter-example must violate at least one expected condition.

How different components are interpreted:

- `deal.pre` is **given**.
- `deal.post` is **expected**.
- `deal.pre` for function called from this function is **expected**.
- `deal.ensure` is **expected**.
- `deal.raises` is **expected** to contain every exception the function can ever raise.
- `assert` is **expected**.

Background

- **1936**. Halting problem. Alan Turing proved that you cannot formally verify if a program will ever finish execution. This is one of the most important theorems of formal verification.
- **1949**. Alan Turing publishes the first ever proof of program correctness.
- **1967**. Robert W. Floyd published "Assigning Meanings to Programs", introducing the idea of program verification using logical assertions.
- **1969**. Hoare logic. Tony Hoare, based on the work of Floyd, introduced the idea of precondition, postcondition, and loop invariant. What's more important, he proved that this is all you need to formally verify correctness of any program. The only thing you can't verify this way is if the program will ever stop.
- **1986**. Design by contract. Bertrand Meyer designed Eiffel programming language which introduced the idea of Design by Contract (DbC). DbC is heavily based on Hoare Logic, but this time it turned from a purely mathematical reasoning into an actual OOP language.
- **2009**. Dafny. Microsoft Research releases a programming language that actually uses Hoare logic to formally prove contracts.
- **2015**. Z3. Microsoft Research opens the source code of the formal verifier used inside of Dafny and in a few other places. Z3 provides bindings for many different programming languages, including Python.
- **2018**. deal. We released a Python library that allows to specify contracts that can be validated in runtime. The initial motivation to make another one DbC library is to provide decorator-based API (as opposed to a more popular docstring-based approach), so users can benefit from syntax-highlighting, autocomplete, autoformatters and other tooling.
- **2019**. Deal gets a linter. It finds contract violations using static analysis.
- **2021**. deal-solver. We released a tool that converts Python code (including deal contracts) into Z3 theorems that can be formally verified.

Limitations

Since Python is a dynamically typed interpreted language and is not designed for formal verification, turning Python code into mathematical expressions is very hard. To name a few limitations:

- There are some mutability bugs that cannot be detected by deal-solver because there is no conception of mutability (and variables) in Z3.
- `set` cannot be converted into `list` because sets are infinite in Z3.
- Some formulas (like `a ** x` where x is a variable) take unreasonable amount of time to prove. To avoid freezing to death, deal-solvers sets a timeout for every proof.
- In general, resolving OOP magic statically is hard. At the moment, deal-solver supports only built-in types and validates only functions and static methods.
- Big chunk of standard library is written on C. So, to support the whole standard library, we have to manually rewrite every function implementation as a Z3 formula (because deal-solver can interpret only Python, not C).
- Verification of loops requires loop invariants. However, deal currently doesn't have such thing because it's not so helpful for other components of the project.

So, deal-solver is more proof-of-concept rather than something that will be a part of your day-to-day tooling.

Further reading

There are few additional links in case if you want to go down the rabbit hole and dig into some hardcore math:

- Engineering Trustworthy Software Systems (do not read this article on sci-hub if you respect paywalls and think that knowledge should be available only for the science elite, piracy is wrong)
- Programming Z3
- Z3Py Guide
- A Tested Semantics for the Python Programming Language
- A Peek Inside SAT Solvers

1.4.11 CrossHair

Warning: CrossHair is an **experimental** tool and it runs your code. So, use it only with safe functions, don't run it on the code that may wipe out your system or do bank transactions.

CrossHair is a third-party tool for finding bugs in Python code with deal support. It is a verifier-driven fuzzer (or "concolic testing"), something in between deal *Tests* and *Formal verification*. It calls the given function multiple times but instead of actual values it passes special mocks, allowing it explore different execution branches.

Installation:

```
python3 -m pip install --user crosshair-tool
```

Usage:

```
python3 -m crosshair watch ./examples/div.py
```

Note: CrossHair is a third-party tool. We're not responsible for bugs in this integration. Use CrossHair issue tracker for all issues you encounter.

Further reading:

- CrossHair documentation
- Deal Support
- How Does It Work?

1.4.12 More on writing contracts

Generating contracts

The best way to get started with deal is to automatically generate some contracts using *decorate* CLI command:

```
python3 -m deal decorate my_project/
```

It will run *Linter* on your code and add some of the missed contracts. The rest of contacts is still on you, though. Also, you should carefully check the generated code for correctness, deal may miss something.

The following contracts are supported by the command and will be added to your code:

- *deal.has*
- *deal.raises*
- *deal.safe*

Simplified signature

The main problem with contracts is that they have to duplicate the original function's signature, including default arguments. While it's not a problem for small examples, things become more complicated when the signature grows. In this case, you can specify a function that accepts only one _ argument, and deal will pass there a container with arguments of the function call, including default ones:

```
@deal.pre(lambda _: _.a + _.b > 0)
def f(a, b=1):
    return a + b

f(1)
# 2

f(-2)
# PreContractError: expected a + b > 0 (where a=-2, b=1)
```

deal.chain

The *deal.chain* decorator allows to merge a few contracts together into one decorator. It can be used to store contracts separately from the function:

```
contract_for_min = deal.chain(
    deal.pre(lambda items: len(items) > 0),
    deal.ensure(lambda items, result: result in items),
)

@contract_for_min
def min(items):
    ...
```

This allows to reuse contracts among multiple functions. Also, it keeps the function signature more clean, multiple decorators may make it a bit noisy.

deal.inherit

The *deal.chain* decorator makes a method to inherit contracts from the base class.

It can be applied to a separate method:

```
class Shape:
    @deal.post(lambda r: r > 0)
    def get_sides(self):
        raise NotImplementedError

class Triangle(Shape):
    @deal.inherit
    def get_sides(self):
        return 3
```

(continues on next page)

(continued from previous page)

```
triangle = Triangle()
triangle.get_sides()
```

Or to the whole class, so all contracts for all methods will be inherited:

```
@deal.inherit
class Line(Shape):
    def get_sides(self):
        return 2

line = Line()
line.get_sides()
# PreContractError: expected r > 0 (where r=2)
```

If the class has multiple base classes, contracts from all of them will be inherited.

If the method already has other contracts or decorators, they will be preserved. Just make sure they all are specified below `@deal.inherit`.

Typing

We encourage you to use type annotations, and so deal is fully type annotated and respects and empowers your type annotations as well. At the same time, deal is very flexible about what can be a validator for a contract (functions, short signatures, Marshmallow schemas etc), and so it cannot be properly described with type annotations. To solve this issue, deal provides a custom plugin for mypy. **The plugin checks types for validators**. It does not execute contracts.

The best way to configure mypy is using `pyproject.toml`:

```
[tool.mypy]
plugins = ["deal.mypy"]
```

Keep in mind that `pyproject.toml` is supported by mypy only starting from version 0.910. Check your installed version by running `mypy --version`. If it is below 0.910, upgrade it by running `python3 -m pip install -U mypy`.

Providing an error

You can provide `message` argument for a contract, and this message will be used as the error message (and in *documentation*):

```
@deal.pre(lambda x: x > 0, message='x must be positive')
def f(x):
    return list(range(x))

f(-2)
# PreContractError: x must be positive (where x=-2)
```

If a single contract includes multiple checks, it can return an error message instead of `False`, so different failures can be distinguished:

```
def contract(x):
    if not isinstance(x, int):
```

(continues on next page)

(continued from previous page)

```
        return 'x must be int'
    if x <= 0:
        return 'x must be positive'
    return True

@deal.pre(contract)
def f(x):
    return list(range(x))

f('Aragorn')
# PreContractError: x must be int (where x='Aragorn')

f(-2)
# PreContractError: x must be positive (where x=-2)
```

External validators

Deal supports a lot of external validation libraries, like Marshmallow, WTForms, PyScheme etc. For example:

```
import deal
import marshmallow

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

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

func('Chris')
'ChrisChris'

func(123)
# PreContractError: [Error(message='Not a valid string.', field='name')] (where␣
↪name=123)
```

See vaa documentation for details.

Performance

Deal tries to be as performant as possible, with the following goals in mind:

- If something can be done only once (in other words, cached) with benefit for performance, must be done only once.
- Heavy operations must not be performed when decorator is just applied, otherwise it negatively affects the import time for the project that uses deal.
- Simplicity must not be sacrificed for performance.

As the outcome, deal has some heavy operations. Namely, introspection of the wrapped function and the validator. THese operations are performed only once, when the function is called in the first time. The idea is similar to how Just-In-Time compilation works in Julia: compile it only when you need it.

So, if you benchmark a function decorated with deal, you can either:

- Disable contracts using `deal.disable`;

- Call the function once in advance to trigger the caching;
- Or pre-cache contracts for a specific function using `deal.introspection.init_all`.

1.4.13 Contracts for modules

The function `module_load` 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.

Example

`__init__.py`:

```
import deal

deal.activate()

from .other import something
```

`other.py`:

```
import deal
import something_else

deal.module_load(deal.pure)

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

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.

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.

1.4.14 Dispatch

> **Warning:** This feature is **experimental**. It works and the API is stable but the behavior in some corner cases may change in the future.

The decorator `deal.dispatch` allows combining multiple implementations of a function into one. When the combined function is called, deal will try to execute every implementation and return the result of the first one that hasn't raised `PreContractError`.

```
@deal.dispatch
def age2stage(age: int) -> str:
    raise NotImplementedError

@age2stage.register
@deal.pre(lambda age: age < 12)
def _(age: int) -> int:
    return 'kid'

@age2stage.register
@deal.pre(lambda age: age < 18)
def _(age: int) -> int:
    return 'teen'

age2stage(10)  # 'kid'
age2stage(14)  # 'teen'
```

If the given arguments passed pre-conditions for none of the implementations, `NoMatchError` is raised:

```
age2stage(20)
# NoMatchError: expected age < 12 (where age=20); expected age < 18 (where age=20)
```

To avoid it, just add a default implementation:

```
@age2stage.register
def _(age: int) -> int:
    return 'adult'

age2stage(20)  # 'adult'
```

The initially decorated function (which you directly pass into `@deal.dispatch`) is never executed. It is used only to provide the name, docstring, and type annotations for the combined function. However, we specify `raise NotImplementedError` instead of just `pass` as the function body, so type checkers won't complain about invalid return type.

Since dispatch requires contracts to be enabled, when you call a dispatched function, contracts get forcefully enabled for the function duration. It may be changed in the future to enable only needed *deal.pre* contracts. So, keep it in mind: if you want to disable all contracts on the production all together, `deal.dispatch` could be a bad fit for your application.

Motivation

The decorator was introduced as a way to do the same as functools.singledispatch but using pre-conditions instead of types. It gives you much more flexibility, allowing you to implement anything you could do in some other languages with the combination of pattern matching, guards, and function overloading. A classic example is recursively calculating factorial.

In Elixir:

```
defmodule Math do
    @spec fac(integer) :: integer
    @doc """
    Calculate factorial of the given number.
    """
    def fac(0), do: 1
    def fac(n) when n > 0, do: n * fac(n - 1)
end

Math.fac(5)    # 120
Math.fac(-1)   # FunctionClauseError
```

And the same in Python using `deal.dispatch`:

```
@deal.dispatch
def fac(n: int) -> int:
    """Calculate factorial of the given number.
    """
    raise NotImplementedError

@fac.register
@deal.pre(lambda n: n == 0)
def _(n):
    return 1

@fac.register
@deal.pre(lambda n: n > 0)
def _(n):
    return n * fac(n - 1)

fac(5)    # 120
fac(-1)   # NoMatchError
```

The implementation based on `deal.dispatch` is more verbose. However, it pays back when each implementation is complex. For example, reading a config in different formats.

Simple solution:

```
from pathlib import Path

def read_config(path: Path) -> Config:
    if path.suffix == '.json':
        ...
    if path.suffix in {'.yml', '.yaml'}:
        ...
    raise ValueError
```

And solution with dispatch:

```
@deal.dispatch
def read_config(path: Path) -> Config:
    raise NotImplementedError

@read_config.register
@deal.pre(lambda path: path.suffix == '.json')
def read_json(path):
    ...

@read_config.register
@deal.pre(lambda path: path.suffix in {'.yml', '.yaml'})
def read_yaml(path):
    ...
```

The latter gives multiple benefits:

1. Each implementation is isolated from others, and so it is easier to read and maintain.
2. Each implementation can be called directly, by users of from tests.
3. Pre-conditions are attached to implementations rather than the `read_config` function, so even if an implementation is called directly, we still can be sure that it is used correctly.
4. Users can register new implementations. So, you have a plugins system out of the box.

1.4.15 Documentation

Sphinx autodoc

Deal has an integration with sphinx documentation generator, namely with autodoc extension. The integration includes all contracts for documented functions into the generated documentation.

The minimal config:

```
import deal

extensions = ['sphinx.ext.autodoc']

def setup(app):
    deal.autodoc(app)
```

And that's all. Now, every time you include something using autodoc, deal will automatically inject documentation for contracts.

This is how deal converts every contract into text:

1. If the contract has `message` argument specified, it is used.
2. If the contract is a named function, the function name is used.
3. If the contract is a lambda, the source code is used.

See also *sphinx* example.

deal.example

The decorator *deal.example* allows to provide a usage example for the decorated function. This example is executed only when running *tests* and partially checked by the linter. It's not, however, executed at runtime. The example must return `True` if it is valid.

```
@deal.example(lambda: double(3) == 6)
def double(x):
    return x * 2
```

Depending on the context and on the mypy version you use, you may encounter `Cannot determine type of "double"` error message from mypy (see mypy#11212). If you do, you can:

1. Upgrade mypy version above 0.910.
2. Add `# type: ignore[has-type]` to the reported line.
3. Add has-type code into disable_error_code list in the mypy configuration file.

If you want to provide an example of when the function raises an exception, you can catch and compare this exception using *deal.catch*:

```
@deal.example(lambda: deal.catch(div, 4, 0) is ZeroDivisionError)
@deal.raises(ZeroDivisionError)
@deal.reason(ZeroDivisionError, lambda x: x == 0)
def div(x, y):
    return x / y
```

For more complex examples (requiring setup, teardown, or complicated arguments) use doctest.

Writing docstrings

The Writing docstrings page of Sphinx documentation provides a good description on how to write docstrings in RST format. Also, there is PEP-257 which describes stylistic conventions related to docstrings (and tells what docstrings are).

Using Markdown

If you prefer more human-friendly Markdown, it needs a bit of hacking. The MyST-Parser extension allows to use Markdown for Sphinx documentation but not for docstrings (see #228). If you want Markdown support for docstrings, you can add m2r2 converter into `docs/conf.py` as a hook for autodoc:

```
from m2r2 import convert

def autodoc_process(app, what, name, obj, options, lines):
    if not lines:
        return lines
    text = convert('\n'.join(lines))
    lines[:] = text.split('\n')

def setup(app):
    app.connect('autodoc-process-docstring', autodoc_process)
```

It doesn't matter what format you choose, deal supports all of them.

1.4.16 Stubs

When *linter* analyses a function, it checks all called functions inside it, even if these functions have no explicit contracts. For example:

```
import deal

def a():
    raise ValueError

@deal.raises(NameError)
def b():
    return a()
```

Here deal finds and error:

```
$ python3 -m deal lint tmp.py
tmp.py
  8:11 raises contract error (ValueError)
    return a()
```

However, in the next case deal doesn't report anything:

```
import deal

def a():
    raise ValueError

def b():
    return a()

@deal.raises(NameError)
def c():
    return b()
```

That's because the exception is raised deep inside the call chain. Analyzing function calls too deep would make deal too slow. The solution is to make contracts for everything in your code that you want to be analyzed. However, when it's about third-party libraries where you can't modify the code, stubs come into play.

Use `stub` command to generate stubs for a Python file:

```
python3 -m deal stub /path/to/a/file.py
```

The command above will produce `/path/to/a/file.json` stub. On the next runs linter will use it do detect contracts.

Built-in stubs

Deal comes with a few pre-generated stubs that are automatically used by the linter:

- Standard library (CPython 3.7)
- marshmallow
- python-dateutil
- pytz
- requests

- urllib3

1.4.17 More on testing

This section assumes that you're familiar with *basic testing* and describes how you can get more from deal testing mechanisms.

Finding memory leaks

Sometimes, when a function is completed, it leaves in memory other objects except result. For example:

```
cache = {}
User = dict

def get_user(name: str) -> User:
    if name not in cache:
        cache[name] = User(name=name)
    return cache[name]
```

Here, `get_user` creates a `User` object and stores it in a global cache. In this case, this "leak" is a desired behavior and we don't want to fight it. This is why we can't a tool (or something right in the Python interpreter) that catches and reports such behavior, it would have too many false-positives.

However, things are different with pure functions. A pure function can't store anything on a side because it is a side effect. The result of a pure function is only what it returns.

The command `memtest` uses this idea to find memory leaks in pure functions. How it works:

1. It finds all pure functions (as `test` does).
2. For every function:
    1. It makes memory snapshot before running the function.
    2. It runs the function with different autogenerated input arguments (as `test` command does) without running contracts and checking the return value type (to avoid side-effects from deal itself).
    3. It makes memory snapshot after running the function.
    4. Snapshots "before" and "after" are comapared. If there is a difference it will be printed.

The return code is equal to the amount of functions with memory leaks.

If the function fails, the command will ignore it and still test the function for leaks. Side-effects shouldn't happen unconditionally, even if the function fails. If you want to find unexpected failures, use `test` command instead.

Constant value for arguments

The `deal.cases` constructor accepts `kwargs` argument where you can specify constant values for the function arguments. For example:

```
@deal.raises(ZeroDivisionError)
def div(a: int, b: int) -> float:
    assert a == 1
    return a / b

# Every test case calls `div` function with `a=1`.
```

(continues on next page)

(continued from previous page)

```
# So, random values are generated only for `b`.
@deal.cases(div, kwargs=dict(a=1))
def test_div(case):
    case()
```

Custom strategies

Under the hood, `deal.cases` uses hypothesis testing framework to generate test cases. The trick is that `kwargs` argument of `deal.cases` can contain hypothesis strategies:

```
import hypothesis.strategies as st

@deal.raises(ZeroDivisionError)
def div(a: int, b: int) -> float:
    assert a >= 10
    return a / b

cases = deal.cases(
    func=div,
    kwargs=dict(
        a=st.integers(min_value=10),
    ),
)
for case in cases:
    case()
```

Reproducible failures

Argument `seed` of `deal.cases` is a random seed. That means, for the same seed value you always get the same test cases. It is a must-have thing for CI. There are a few important things:

- If the seed is different for diferrent pipelines, it will run a bit different test cases every time which increases your chances to find a tricky corner case.
- If the seed is the same when you re-run the same job, it will help you to identify flaky tests. In other words, if a CI job fails, you re-run it, and it passes, it was a flaky test which happens because of tricky side-effects (database connection failed, another test changed a state etc.) and it will be hard to reproduce. However, if re-run fails with the same error, most probably it is a failure that you can easily reproduce and debug locally, knowing the seed.
- The seed should be shown in the CI job to make it possible to use it locally to reproduce a failure. It is either printed in the job output or is something known like the pipeline run number.

There is an example for GitLab CI:

```
import os
import deal

seed = None
if os.environ.get('CI_PIPELINE_ID'):
    seed = int(os.environ['CI_PIPELINE_ID'])

@deal.cases(div, seed=seed)
def test_div(case):
    case()
```

Fuzzing

Fuzzer is when an external tool or library (fuzzer) generates a bunch of random data in hope to break your program. That means, fuzzing requires a lot of resources and is performance-critical. This is why most of the fuzzers are written on C. However, there are few Python wrappers for existing fuzzers to simplify fuzzing for Python functions:

- atheris
- python-afl
- pythonfuzz

The `deal.cases` object can be used as a target for any fuzzer.

Atheris:

```
import atheris

test = deal.cases(div)
atheris.Setup([], test)
atheris.Fuzz()
```

PythonFuzz:

```
from pythonfuzz.main import PythonFuzz

test = deal.cases(div)
PythonFuzz(test)()
```

See *fuzzing_atheris* and *fuzzing_pythonfuzz* examples for the full code.

Iteration over cases

The `deal.cases` object can be used not only as a function or decorator but also as an iterable. On iteration, it emits the test cases, so you can have more control over what and when to run:

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

However, in this case deal doesn't know which cases have failed and can't provide that information back into hypothesis for shrinking (finding the smallest example to reproduce a failure). So while it is the same as decorator when everything is fine, it will provide a bit uglier report on failure.

Mixing with Hypothesis

Under the hood, deal uses hypothesis to generate test cases. So, we can mix `deal.cases` with hypothesis decorators. The only exception is `hypothesis.settings` which should be passed into `deal.cases` as `settings` argument because hypothesis doesn't support application of settings twice but deal applies its own default settings.

See *using_hypothesis* example.

1.4.18 Recipes

Some ideas that are useful in the real world applications.

Keep contracts simple

If a function accepts only a few short arguments, duplicate the original signature (without annotations) for contracts:

```
@deal.pre(lambda left, right: right != 0)
def div(left: float, right: float) -> float:
    return left / right
```

Otherwise, or if a function has default arguments, use simplified signature for contracts:

```
@deal.pre(lambda _: _.default is not None or _.right != 0)
def div(left: float, right: float, default: float = None) -> float:
    try:
        return left / right
    except ZeroDivisionError:
        if default is not None:
            return default
        raise
```

Don't check types

Never check types with deal. MyPy does it much better. Also, there are plenty of alternatives for both static and dynamic validation. Deal is intended to empower types, to tell a bit more about possible values set than you can do with type annotations, not replace them. However, if you want to play with deal a bit or make types a part of contracts, PySchemes-based contract is a good solution:

```
import deal
from pyschemes import Scheme

@deal.pre(Scheme(dict(left=str, right=str)))
def concat(left, right):
    return left + right

concat('ab', 'cd')
# 'abcd'

concat(1, 2)
# PreContractError: at key 'left' (expected type: 'str', got 'int')
```

Prefer `pre` and `post` over `ensure`

If a contract needs only function arguments, use `pre`. If a contract checks only function result, use `post`. And only if a contract need both input and output values at the same time, use `ensure`. Keeping available namespace for contract as small as possible makes the contract signature simpler and helps with partial execution in the linter.

Prefer `reason` over `raises`

Always try your best to tell why exception can be raised. However, keep in mind that all exceptions from `reason` still have to be explicitly specified in `raises` since contracts are isolated and have no way to exchange information between each other:

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

Keep module initialization pure

Nothing should happen on module load. Create some constants, compile RegExes, and that's all. Make it lazy.

```
deal.module_load(deal.pure)
```

Contracts shouldn't be important

Never catch contract errors. Never rely on them in runtime. They are for tests and humans. The shouldn't have an actual logic, only validate it.

Short signature conflicts

In short signature, _ is a `dict` with access by attributes. Hence it has all dict attributes. So, if argument we need conflicts with a dict attribute, use getitem instead of getattr. For example, we should use `_['items']` instead of `_.items`.

Keep contracts pure

You can use any logic inside the validator. However, thumb up rule is to keep contracts pure (without any side-effects, even logging). The main motivation for it is that some contracts can be partially executed by *linter*.

The `message` is description, not error

The `message` argument should tell what is expected behavior without assuming that the user violated it. This is because the users can encounter it not only when a `ContractError` is raised but also when they just read the source code or *generated documentation*. For example, if your contract checks that `b >= 0`, don't say “b is negative” (what is violated), say “b must be non-negative” (what is expected).

Permissive license

Deal is distributed under MIT License which is a permissive license with high license compatibility. However, Deal has astroid in the dependencies which is licensed under LGPL. While this license allows to be used in non-LGPL proprietary software too, it still can be not enough for some companies. So, if the legal department in your company forbids using LGPL libraries in transitive dependencies, you can freely remove `astroid` from the project dependencies before shipping it on the production. All CLI commands won't work anymore but runtime checks will.

1.4.19 Examples

choice

```python
import random
from typing import List

import deal


# the list cannot be empty
@deal.pre(lambda items: bool(items))
# result is an element withit the given list
@deal.ensure(lambda items, result: result in items)
@deal.has('random')
def choice(items: List[str]) -> str:
    """Get a random element from the given list.
    """
    return random.choice(items)


test_choice = deal.cases(choice)
```

concat

```python
import deal


@deal.ensure(lambda _: _.result.startswith(_.left))
@deal.ensure(lambda _: _.result.endswith(_.right))
@deal.ensure(lambda _: len(_.result) == len(_.left) + len(_.right))
@deal.has()
def concat(left: str, right: str) -> str:
    """Concatenate 2 given strings.

    https://deal.readthedocs.io/basic/motivation.html
    """
    return left + right


test_concat = deal.cases(concat)
```

count

```python
from typing import List

import deal


# In short signature, `_` is a `dict` with access by attributes.
# Hence it has all dict attributes. So, if argument we need conflicts
# with a dict attribute, use getitem instead of getattr.
# In the example below, we use `_['items']` instead of `_.items`.
```

(continues on next page)

(continued from previous page)

```
@deal.post(lambda result: result >= 0)
# if count is not zero, `item` appears in `items` at least once.
@deal.ensure(lambda _: _.result == 0 or _['item'] in _['items'])
# if count is zero, `item` is not in `items`
@deal.ensure(lambda _: _.result != 0 or _['item'] not in _['items'])
@deal.has()
def count(items: List[str], item: str) -> int:
    """How many times `item` appears in `items`
    """
    return items.count(item)


test_count = deal.cases(count)
```

div

```
import deal


@deal.raises(ZeroDivisionError)
@deal.reason(ZeroDivisionError, lambda _: _.right == 0)
@deal.has()
def div1(left: float, right: float) -> float:
    """
    This implementation allows zero to be passed
    but raises ZeroDivisionError in that case.
    """
    return left / right


@deal.pre(lambda _: _.right != 0)
@deal.has()
def div2(left: float, right: float) -> float:
    """
    This implementation doesn't allow zero to be passed in a function.
    If it is accidentally passed, PreConditionError will be raised
    and the funcrion won't be executed.
    """
    return left / right


test_div1 = deal.cases(div1)
test_div2 = deal.cases(div2)
```

fuzzing_atheris

```python
"""
Get help for libFuzzer:
    python3 examples/fuzzing_atheris.py -help=1

Run 1000 test cases:
    python3 examples/fuzzing_atheris.py -runs=1000
"""
import codecs
import sys

import atheris

import deal


def encode(text: str) -> str:
    return codecs.encode(text, encoding='rot13')


@deal.ensure(lambda text, result: encode(result) == text)
def decode(text: str) -> str:
    assert text != 'bad'
    return codecs.encode(text, encoding='rot13')


def fuzz():
    test = deal.cases(decode)
    atheris.Setup(sys.argv, test)
    atheris.Fuzz()


if __name__ == '__main__':
    fuzz()
```

fuzzing_pythonfuzz

```python
import codecs

from pythonfuzz.main import PythonFuzz

import deal


def encode(text: str) -> str:
    return codecs.encode(text, encoding='rot13')


@deal.ensure(lambda text, result: encode(result) == text)
def decode(text: str) -> str:
    assert text != 'bad'
    return codecs.encode(text, encoding='rot13')


def fuzz():
```

(continues on next page)

(continued from previous page)

```
    test = deal.cases(decode)
    PythonFuzz(test)()


if __name__ == '__main__':
    fuzz()
```

using_hypothesis

```
from typing import List

import hypothesis

import deal


@deal.pre(lambda items: len(items) > 0)
@deal.has()
def my_min(items: List[int]) -> int:
    return min(items)


@hypothesis.example([1, 2, 3])
@deal.cases(
    func=my_min,
    settings=hypothesis.settings(
        verbosity=hypothesis.Verbosity.normal,
    ),
)
def test_min(case):
    case()


if __name__ == '__main__':
    test_min()
```

index_of

```
from typing import List

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,
```

(continues on next page)

(continued from previous page)

```
        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),
    # no side-effects
    deal.has(),
)


@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


test_index_of = deal.cases(index_of)
```

min

```
from typing import List

import deal


@deal.pre(lambda items: len(items) > 0)
@deal.has()
def my_min(items: List[int]) -> int:
    return min(items)


@deal.has('stdout')
def example():
    # good
    print(my_min([3, 1, 4]))
    # bad
    print(my_min([]))


test_min = deal.cases(my_min)
```

Linter output:

```
$ python3 -m deal lint examples/min.py
examples/min.py
  21:4 DEAL011 pre contract error ([])
    my_min([])
    ^
```

format

```python
import re

import deal


def contract(template: str, *args):
    rex = re.compile(r'\{\:([a-z])\}')
    types = {'s': str, 'd': float}
    matches = rex.findall(template)
    if len(matches) != len(args):
        return f'expected {len(matches)} argument(s) but {len(args)} found'
    for match, arg in zip(matches, args):
        t = types[match[0]]
        if not isinstance(arg, t):
            return f'expected {t.__name__}, {type(arg).__name__} given'
    return True


@deal.pre(contract)
def format(template: str, *args) -> str:
    return template.format(*args)


@deal.has('io')
def example():
    # good
    print(format('{:s}', 'hello'))

    # bad
    print(format('{:s}'))                # not enough args
    print(format('{:s}', 'a', 'b'))      # too many args
    print(format('{:d}', 'a'))           # bad type


if __name__ == '__main__':
    print(format('{:s} {:s}', 'hello', 'world'))
```

Linter output:

```
$ python3 -m deal lint examples/format.py
examples/format.py
  32:10 DEAL011 expected 1 argument(s) but 0 found ('{:s}')
    print(format('{:s}'))                # not enough args
          ^
  33:10 DEAL011 expected 1 argument(s) but 2 found ('{:s}', 'a', 'b')
    print(format('{:s}', 'a', 'b'))      # too many args
          ^
  34:10 DEAL011 expected float, str given ('{:d}', 'a')
    print(format('{:d}', 'a'))           # bad type
          ^
```

sphinx

Source code:

```python
import deal


@deal.reason(ZeroDivisionError, lambda a, b: b == 0)
@deal.reason(ValueError, lambda a, b: a == b, message='a is equal to b')
@deal.raises(ValueError, IndexError, ZeroDivisionError)
@deal.pre(lambda a, b: b != 0)
@deal.pre(lambda a, b: b != 0, message='b is not zero')
@deal.ensure(lambda a, b, result: b != result)
@deal.post(lambda res: res != .13)
@deal.has('database', 'network')
@deal.example(lambda: example(6, 2) == 3)
def example(a: int, b: int) -> float:
    """Example function.

    :return: The description for return value.
    """
    return a / b
```

Sphinx config (`docs/conf.py`):

```python
import deal

extensions = ['sphinx.ext.autodoc']

def setup(app):
    deal.autodoc(app)
```

Including into a documentation page (`docs/index.rst`):

```rst
.. autofunction:: examples.sphinx.example
```

Generated output:

`examples.sphinx.`**`example`**(*a: int*, *b: int*) → float

Example function.

Returns The description for return value.

Side-effects

- network
- database

Raises

- **`IndexError`** –
- **`ValueError`** – a is equal to b
- **`ZeroDivisionError`** – `b == 0`

Contracts

- b is not zero
- `b != 0`

- `res != .13`
- `b != result`

Examples

- `example(6, 2) == 3`

1.4.20 CLI

lint

`deal._cli._lint.`**`LintCommand`**(*stream: TextIO*, *root: pathlib.Path*) → None

Run linter against the given files.

```
python3 -m deal lint project/
```

Options:

- `--json`: output violations as json per line.
- `--nocolor`: output violations in human-friendly format but without colors. Useful for running linter on CI.

Exit code is equal to the found violations count. See linter documentation for more details.

decorate

`deal._cli._decorate.`**`DecorateCommand`**(*stream: TextIO*, *root: pathlib.Path*) → None

Add decorators to your code.

```
python3 -m deal decorate project/
```

Options:

- `--types`: types of decorators to apply. All are enabled by default.
- `--double-quotes`: use double quotes. Single quotes are used by default.
- `--nocolor`: do not use colors in the console output.

The exit code is always 0. If you want to test the code for missed decorators, use the `lint` command instead.

test

`deal._cli._test.`**`TestCommand`**(*stream: TextIO*, *root: pathlib.Path*) → None

Generate and run tests against pure functions.

```
python3 -m deal test project/
```

Function must be decorated by one of the following to be run:

- `@deal.pure`
- `@deal.has()` (without arguments)

Options:

- `--count`: how many input values combinations should be checked.

Exit code is equal to count of failed test cases. See tests documentation for more details.

memtest

`deal._cli._memtest.`**`MemtestCommand`**(*stream: TextIO*, *root: pathlib.Path*) → None

Generate and run tests against pure functions and report memory leaks.

```
python3 -m deal memtest project/
```

Function must be decorated by one of the following to be run:

- `@deal.pure`
- `@deal.has()` (without arguments)

Options:

- `--count`: how many input values combinations should be checked.

Exit code is equal to count of leaked functions. See memory leaks documentation for more details.

stub

`deal._cli._stub.`**`StubCommand`**(*stream: TextIO*, *root: pathlib.Path*) → None

Generate stub files for the given Python files.

```
python3 -m deal stub project/
```

Options:

- `--iterations`: how many time run stub generation against files. Every new iteration uses results from the previous ones, improving the result. Default: 1.

Exit code is 0. See stubs documentation for more details.

prove

`deal._cli._prove.`**`ProveCommand`**(*stream: TextIO*, *root: pathlib.Path*) → None

Verify correctness of code.

```
python3 -m deal prove project/
```

Options:

- `--skipped`: show skipped functions.
- `--nocolor`: disable colored output.

Exit code is equal to the failed theorems count. See Formal Verification documentation for more information.

1.4.21 API

Values

deal.**pre**(*validator*, **, message: Optional[str] = None, exception: Optional[Exception] = None*) → Callable[[C], C]

Decorator implementing precondition value contract.

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

Parameters

- **validator** – a function or validator that implements the contract.
- **message** – error message for the exception raised on contract violation. No error message by default.
- **exception** – exception type to raise on the contract violation. `PreContractError` by default.

Returns a function wrapper.

```
>>> import deal
>>> @deal.pre(lambda a, b: a + b > 0)
... def example(a, b):
...     return (a + b) * 2
>>> example(1, 2)
6
>>> example(1, -2)
Traceback (most recent call last):
  ...
PreContractError: expected a + b > 0 (where a=1, b=-2)
```

deal.**post**(*validator*, **, message: Optional[str] = None, exception: Optional[Exception] = None*) → Callable[[C], C]

Decorator implementing postcondition value contract.

Postcondition is a condition that must be true for the function result. Raises `PostContractError` otherwise.

Parameters

- **validator** – a function or validator that implements the contract.
- **message** – error message for the exception raised on contract violation. No error message by default.
- **exception** – exception type to raise on the contract violation. `PostContractError` by default.

Returns a function wrapper.

```
>>> import deal
>>> @deal.post(lambda res: res > 0)
... def example(a, b):
...     return a + b
>>> example(-1, 2)
1
>>> example(1, -2)
Traceback (most recent call last):
  ...
PostContractError: expected res > 0 (where res=-1)
```

`deal.ensure` (*validator*, *, *message: Optional[str] = None*, *exception: Optional[Exception] = None*) → Callable[[C], C]

Decorator implementing postcondition value contract.

Postcondition is a condition that must be true for the function result. Raises `PostContractError` otherwise. It's like *@deal.post* but contract accepts as input value not only the function result but also the function input arguments. The function result is passed into validator as `result` keyword argument.

Parameters

- **validator** – a function or validator that implements the contract.
- **message** – error message for the exception raised on contract violation. No error message by default.
- **exception** – exception type to raise on the contract violation. `PostContractError` by default.

Returns a function wrapper.

```
>>> import deal
>>> @deal.ensure(lambda a, result: a < result)
... def example(a):
...     return a * 2
>>> example(2)
4
>>> example(0)
Traceback (most recent call last):
  ...
PostContractError: expected a < result (where result=0, a=0)
```

`deal.inv` (*validator*, *, *message: Optional[str] = None*, *exception: Optional[Exception] = None*) → Callable[[T], T]

Decorator implementing invariant value contract.

Invariant is a condition that can be relied upon to be true during execution of a program. `@deal.inv` is triggered in 3 cases:

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

Deal doesn't rollback changes on contract violation.

Parameters

- **validator** – a function or validator that implements the contract.
- **message** – error message for the exception raised on contract violation. No error message by default.
- **exception** – exception type to raise on the contract violation. `InvContractError` by default.

Returns a class wrapper.

```
>>> import deal
>>> @deal.inv(lambda obj: obj.likes >= 0)
... class Video:
...   likes = 1
...   def like(self): self.likes += 1
```

(continues on next page)

(continued from previous page)

```
...     def dislike(self): self.likes -= 1
...
>>> v = Video()
>>> v.dislike()
>>> v.likes
0
>>> v.dislike()
Traceback (most recent call last):
...
InvContractError: expected obj.likes >= 0
>>> v.likes
-1
>>> v.likes = 2
>>> v.likes = -2
Traceback (most recent call last):
...
InvContractError: expected obj.likes >= 0
>>> v.likes
-2
```

`deal.example`(*validator: Callable[], bool]*) → Callable[[C], C]

Decorator for providing a usage example for the wrapped function.

The example isn't checked at runtime. Instead, it is run in tests and checked by the linter. The example should use the decorated function and return True if the result is expected.

```
>>> import deal
>>> @deal.example(lambda: double(3) == 6)
... def double(x):
...     return x * 2
...
```

`deal.dispatch`(*func: C*) → deal._runtime._dispatch.Dispatch[C]

Combine multiple functions into one.

When the decorated function is called, it will try to call all registered functions and return the result from the first one that doesn't raise `PreContractError`.

```
>>> import deal
>>> @deal.dispatch
... def double(x: int) -> int:
...     raise NotImplementedError
...
>>> @double.register
... @deal.pre(lambda x: x == 3)
... def _(x: int) -> int:
...     return 6
...
>>> @double.register
... @deal.pre(lambda x: x == 4)
... def _(x: int) -> int:
...     return 8
...
>>> double(3)
6
>>> double(4)
8
```

(continues on next page)

(continued from previous page)

```
>>> double(5)
Traceback (most recent call last):
    ...
NoMatchError: expected x == 3 (where x=5); expected x == 4 (where x=5)
```

Side-effects and exceptions

`deal.has`(**markers: str, message: Optional[str] = None, exception: Optional[Exception] = None*) → Callable[[C], C]

Decorator controlling side-effects of the function.

Allows to specify markers identifying which side-effect the functon may have. Side-effects must be propagated into all callers using `deal.has`, this is controlled by the linter. Some side-effects are also checked at runtime.

```
>>> import deal
>>> @deal.has()
... def greet():
...   print('hello')
...
>>> greet()
Traceback (most recent call last):
    ...
SilentContractError
>>> @deal.has('stdout')
... def greet():
...   print('hello')
...
>>> greet()
hello
```

`deal.raises`(**exceptions: Type[Exception], message: Optional[str] = None, exception: Optional[Exception] = None*) → Callable[[C], C]

Decorator listing the exceptions which the function can raise.

Implements exception contract. If the function raises an exception not listed in the decorator, `RaisesContractError` will be raised.

Parameters

- **exceptions** – exceptions which the function can raise.
- **message** – error message for the exception raised on contract violation. No error message by default.
- **exception** – exception type to raise on the contract violation. `RaisesContractError` by default.

Returns a function wrapper.

```
>>> import deal
>>> @deal.raises(ZeroDivisionError, ValueError)
... def div(a, b):
...   return a / b
...
>>> div(1, 0)
Traceback (most recent call last):
    ...
```

(continues on next page)

(continued from previous page)

```
ZeroDivisionError: division by zero
>>> div(1, '')
Traceback (most recent call last):
    ...
 TypeError: unsupported operand type(s) for /: 'int' and 'str'
 The above exception was the direct cause of the following exception:
    ...
RaisesContractError
```

`deal.`**`reason`**(*event: Type[Exception]*, *validator*, ***, *message: Optional[str] = None*, *exception: Optional[Exception] = None*) → Callable[[C], C]

Decorator implementing exception contract.

Allows to assert precondition for raised exception. It's like *@deal.ensure* but when instead of returning result the function raises an exception.

Parameters

- **`event`** – exception raising which will trigger contract validation.
- **`validator`** – a function or validator that implements the contract.
- **`message`** – error message for the exception raised on contract violation. No error message by default.
- **`exception`** – exception type to raise on the contract violation. `ReasonContractError` by default.

Returns a function wrapper.

```
>>> import deal
>>> @deal.reason(ZeroDivisionError, lambda a, b: b == 0)
... def div(a, b):
...   return a / (a - b)
>>> div(2, 1)
2.0
>>> div(0, 0)
Traceback (most recent call last):
    ...
ZeroDivisionError: division by zero
>>> div(2, 2)
Traceback (most recent call last):
    ...
 ZeroDivisionError: division by zero
 The above exception was the direct cause of the following exception:
    ...
ReasonContractError: expected b == 0 (where a=2, b=2)
```

Helpers

deal.**inherit**(*func: TF*) → TF

Inherit contracts from base classes.

Can be used to decorate either the whole class or a separate method.

```
>>> import deal
>>> class Shape:
...   @deal.post(lambda r: r > 2)
...   def get_sides(self):
...     raise NotImplementedError
...
>>> class Triangle(Shape):
...   @deal.inherit
...   def get_sides(self):
...     return 3
...
>>> class Line(Shape):
...   @deal.inherit
...   def get_sides(self):
...     return 2
...
>>> triangle = Triangle()
>>> triangle.get_sides()
3
>>> line = Line()
>>> line.get_sides()
Traceback (most recent call last):
    ...
PreContractError: expected r > 0 (where r=2)
```

deal.**chain**(**contracts: Callable[[C], C]*) → Callable[[F], F]

Decorator to chain 2 or more contracts together.

It can be helpful to store contracts separately from the function. Consider using it when you have too many contracts. Otherwise, the function will be lost under a bunch of decorators.

```
>>> import deal
>>> sum_contract = deal.chain(
...   deal.pre(lambda a, b: a > 0),
...   deal.pre(lambda a, b: b > 0),
...   deal.post(lambda res: res > 0),
... )
>>> @sum_contract
... def sum(a, b):
...   return a + b
...
>>> sum(2, 3)
5
>>> sum(2, -3)
Traceback (most recent call last):
    ...
PreContractError: expected b > 0 (where a=2, b=-3)
>>> sum(-2, 3)
Traceback (most recent call last):
    ...
PreContractError: expected a > 0 (where a=-2, b=3)
```

Parameters **contracts** – contracts to chain.

Returns a function wrapper

deal.**pure** (*_func: C*) → C

Decorator for pure functions.

Alias for `@deal.chain(deal.has(), deal.safe)`.

Pure function has no side-effects and doesn't raise any exceptions.

```
>>> import deal
>>> @deal.pure
... def div(a, b, log=False):
...   if log:
...     print('div called')
...   return a / b
...
>>> div(2, 4)
0.5
>>> div(2, 0)
Traceback (most recent call last):
    ...
 ZeroDivisionError: division by zero
 The above exception was the direct cause of the following exception:
    ...
RaisesContractError
>>> div(2, 3, log=True)
Traceback (most recent call last):
    ...
SilentContractError
```

deal.**safe** (*, *message: str = 'None', exception: Union[Exception, Type[Exception]] = 'None'*) → Callable[[C], C]

deal.**safe** (*_func: C*) → C

Alias for *@deal.raises()*. Wraps a function that never raises an exception.

```
>>> import deal
>>> @deal.safe
... def div(a, b):
...   return a / b
...
>>> div(2, 4)
0.5
>>> div(2, 0)
Traceback (most recent call last):
    ...
 ZeroDivisionError: division by zero
 The above exception was the direct cause of the following exception:
    ...
RaisesContractError
```

deal.**implies** (*test, then: T*) → Union[bool, T]

Check `then` only if `test` is true.

A convenient helper for contracts that must be checked only for some cases. It is known as "implication" or material conditional.

```
>>> import deal
>>> deal.implies(False, False)
True
>>> deal.implies(False, True)
True
>>> deal.implies(True, False)
False
>>> deal.implies(True, True)
True
```

`deal.catch`(*func: Callable*, **args*, ***kwargs*) → Optional[Type[Exception]]

Call the function with the given arguments, catching any exception.

The catched exception is returned. This function may be useful in combination with {py:func}`deal.example`.

```
>>> import deal
>>> @deal.example(lambda: deal.catch(div, 4, 0) is ZeroDivisionError)
... @deal.raises(ZeroDivisionError)
... @deal.reason(ZeroDivisionError, lambda x: x == 0)
... def div(x, y):
...   return x / y
...
>>>
```

Testing

Keep in mind that sphinx skipped some of the docstrings for `deal.cases`.

class `deal.cases`(*func: Callable*, ***, *count: int = 50*, *kwargs: Optional[Dict[str, Any]] = None*, *check_types: bool = True*, *settings: Optional[hypothesis.settings] = None*, *seed: Optional[int] = None*)

Generate test cases for the given function.

`__call__`(*test_func: Callable[[...], None]*) → Callable[[...], None]

`__call__`() → None

`__call__`(*buffer: Union[bytes, bytearray, memoryview, BinaryIO]*) → Optional[bytes]

Allows deal.cases to be used as decorator, test function, or fuzzing target.

`__init__`(*func: Callable*, ***, *count: int = 50*, *kwargs: Optional[Dict[str, Any]] = None*, *check_types: bool = True*, *settings: Optional[hypothesis.settings] = None*, *seed: Optional[int] = None*) → None

Create test cases generator.

```
>>> import deal
>>> @deal.pre(lambda a, b: b != 0)
... def div(a: int, b: int) -> float:
...   return a / b
...
>>> cases = deal.cases(div)
>>>
```

`__iter__`() → Iterator[deal._testing.TestCase]

Emits test cases.

It can be helpful when you want to see what test cases are generated. The recommend way is to use `deal.cases` as a decorator instead.

```
>>> import deal
>>> @deal.pre(lambda a, b: b != 0)
... def div(a: int, b: int) -> float:
...     return a / b
...
>>> cases = iter(deal.cases(div))
>>> next(cases)
TestCase(args=(), kwargs=..., func=<function div ...>, exceptions=(), check_
↪types=True)
>>> for case in cases:
...     result = case()  # execute the test case
>>>
```

`__repr__`() → str
Return repr(self).

`__weakref__`
list of weak references to the object (if defined)

`check_types: bool`
check that the result matches return type of the function. Enabled by default.

`count: int`
how many test cases to generate, defaults to 50.

`func: Callable`
the function to test. Should be type annotated.

`kwargs: Dict[str, Any]`
keyword arguments to pass into the function.

`seed: Optional[int]`
Random seed to use when generating test cases. Use it to make tests deterministic.

`settings: hypothesis.settings`
Hypothesis settings to use instead of default ones.

class deal.**TestCase**(*args: Tuple[Any, ...], kwargs: Dict[str, Any], func: Callable, exceptions: Tuple[Type[Exception], ...], check_types: bool*)
A callable object, wrapper around a function that must be tested.

When called, calls the wrapped function, suppresses expected exceptions, checks the type of the result, and returns it.

property args
Positional arguments to be passed in the function

property check_types
Check that the result matches return type of the function.

property exceptions
Exceptions that must be suppressed.

property func
The function which will be called when the test case is called

property kwargs
Keyword arguments to be passed in the function

Introspection

The module provides `get_contracts` function which enumerates contracts wrapping the given function. Every contract is returned in wrapper providing a stable interface.

Usage example:

```
import deal

@deal.pre(lambda x: x > 0)
def f(x):
    return x + 1

contracts = deal.introspection.get_contracts(f)
for contract in contracts:
    assert isinstance(contract, deal.introspection.Contract)
    assert isinstance(contract, deal.introspection.Pre)
    assert contract.source == 'x > 0'
    assert contract.exception is deal.PreContractError
    contract.validate(1)
```

State management

`deal.`**`disable`**`()` → None
Disable all contracts.

`deal.`**`enable`**`()` → None
Enable all contracts.

`deal.`**`reset`**`()` → None
Restore contracts switch to default.

All contracts are disabled on production by default. See runtime documentation.

Other functions

`deal.`**`autodoc`**`(`*app: SphinxApp*`)` → None
Activate the hook for sphinx that includes contracts into documentation generated by autodoc.

`deal.`**`activate`**`()` → bool
Activate module-level checks.

This function must be called before importing anything with *deal.module_load* contract. Otherwise, the contract won't be executed.

The best practice is to place it in `__init__.py` of your project:

```
>>> import deal
>>> deal.activate()
```

See Contracts for importing modules documentation for more details.

`deal.`**`module_load`**`(`**contracts*`)` → None
Specify contracts that will be checked at module import time. Keep in mind that *deal.activate* must be called before importing a module with `module_load` contract.

```
>>> import deal
>>> deal.module_load(deal.has(), deal.safe)
```

See Contracts for importing modules documentation for more details.

Exceptions

exception deal.**ContractError**(*message: str = ''*, *errors=None*, *validator=None*, *params: Optional[Dict[str, Any]] = None*, *origin: Optional[object] = None*)
The base class for all errors raised by deal contracts.

exception deal.**ExampleContractError**(*message: str = ''*, *errors=None*, *validator=None*, *params: Optional[Dict[str, Any]] = None*, *origin: Optional[object] = None*)
The error raised by `deal.example` for contract violation.

`deal.example` contracts are checked only during testing and linting, not at runtime.

exception deal.**InvContractError**(*message: str = ''*, *errors=None*, *validator=None*, *params: Optional[Dict[str, Any]] = None*, *origin: Optional[object] = None*)
The error raised by `deal.inv` for contract violation.

exception deal.**MarkerError**(*message: str = ''*, *errors=None*, *validator=None*, *params: Optional[Dict[str, Any]] = None*, *origin: Optional[object] = None*)
The base class for errors raised by `deal.has` for contract violation.

exception deal.**NoMatchError**(*exceptions: Tuple[deal.PreContractError, ...]*)
The error raised by `deal.dispatch` when there is no matching implementation.

"No matching implementation" means that all registered functions raised `PreContractError`.

exception deal.**OfflineContractError**(*message: str = ''*, *errors=None*, *validator=None*, *params: Optional[Dict[str, Any]] = None*, *origin: Optional[object] = None*)
The error raised by `deal.has` for networking markers violation.

The networking can be allowed by markers `io`, `network`, and `socket`.

exception deal.**PostContractError**(*message: str = ''*, *errors=None*, *validator=None*, *params: Optional[Dict[str, Any]] = None*, *origin: Optional[object] = None*)
The error raised by `deal.post` for contract violation.

exception deal.**PreContractError**(*message: str = ''*, *errors=None*, *validator=None*, *params: Optional[Dict[str, Any]] = None*, *origin: Optional[object] = None*)
The error raised by `deal.pre` for contract violation.

exception deal.**RaisesContractError**(*message: str = ''*, *errors=None*, *validator=None*, *params: Optional[Dict[str, Any]] = None*, *origin: Optional[object] = None*)
The error raised by `deal.raises` for contract violation.

exception deal.**ReasonContractError**(*message: str = ''*, *errors=None*, *validator=None*, *params: Optional[Dict[str, Any]] = None*, *origin: Optional[object] = None*)
The error raised by `deal.reason` for contract violation.

exception deal.**SilentContractError**(*message: str = ''*, *errors=None*, *validator=None*, *params: Optional[Dict[str, Any]] = None*, *origin: Optional[object] = None*)
The error raised by `deal.has` for printing markers violation.

The printing can be allowed by markers `io`, `print`, `stdout`, and `stderr`.

PYTHON MODULE INDEX

d

deal.introspection, 57

INDEX

Symbols

__call__() (*deal.cases method*), 55
__init__() (*deal.cases method*), 55
__iter__() (*deal.cases method*), 55
__repr__() (*deal.cases method*), 56
__weakref__ (*deal.cases attribute*), 56

A

activate() (*in module deal*), 57
args() (*deal.TestCase property*), 56
autodoc() (*in module deal*), 57

C

cases (*class in deal*), 55
catch() (*in module deal*), 55
chain() (*in module deal*), 53
check_types (*deal.cases attribute*), 56
check_types() (*deal.TestCase property*), 56
ContractError, 58
count (*deal.cases attribute*), 56

D

deal.introspection
    module, 57
DecorateCommand() (*in module deal._cli._decorate*), 46
disable() (*in module deal*), 57
dispatch() (*in module deal*), 50

E

enable() (*in module deal*), 57
ensure() (*in module deal*), 48
example() (*in module deal*), 50
example() (*in module examples.sphinx*), 45
ExampleContractError, 58
exceptions() (*deal.TestCase property*), 56

F

func (*deal.cases attribute*), 56
func() (*deal.TestCase property*), 56

H

has() (*in module deal*), 51

I

implies() (*in module deal*), 54
inherit() (*in module deal*), 53
inv() (*in module deal*), 49
InvContractError, 58

K

kwargs (*deal.cases attribute*), 56
kwargs() (*deal.TestCase property*), 56

L

LintCommand() (*in module deal._cli._lint*), 46

M

MarkerError, 58
MemtestCommand() (*in module deal._cli._memtest*), 47
module
    deal.introspection, 57
module_load() (*in module deal*), 57

N

NoMatchError, 58

O

OfflineContractError, 58

P

post() (*in module deal*), 48
PostContractError, 58
pre() (*in module deal*), 48
PreContractError, 58
ProveCommand() (*in module deal._cli._prove*), 47
pure() (*in module deal*), 54

R

raises() (*in module deal*), 51
RaisesContractError, 58

reason() (*in module deal*), 52
ReasonContractError, 58
reset() (*in module deal*), 57

S

safe() (*in module deal*), 54
seed (*deal.cases attribute*), 56
settings (*deal.cases attribute*), 56
SilentContractError, 58
StubCommand() (*in module deal._cli._stub*), 47

T

TestCase (*class in deal*), 56
TestCommand() (*in module deal._cli._test*), 46