icontract Documentation

Release 2.6.6

Marko Ristin

Jan 07, 2024

CONTENTS:

CHAPTER

ONE

INTRODUCTION

Icontract provides design-by-contract to Python3 with informative violation messages and inheritance.

It also gives a base for a flourishing of a wider ecosystem:

- A linter pyicontract-lint,
- A sphinx plug-in sphinx-icontract,
- A tool icontract-hypothesis for automated testing and ghostwriting test files which infers Hypothesis strategies based on the contracts,
  - together with IDE integrations such as icontract-hypothesis-vim, icontract-hypothesis-pycharm, and icontract-hypothesis-vscode,
- Directly integrated into CrossHair, a tool for automatic verification of Python programs,
  - together with IDE integrations such as crosshair-pycharm and crosshair-vscode, and
- An integration with FastAPI through fastapi-icontract to enforce contracts on your HTTP API and display them in OpenAPI 3 schema and Swagger UI, and
- An extensive corpus, Python-by-contract corpus, of Python programs annotated with contracts for educational, testing and research purposes.

1.1 Related Projects

There exist a couple of contract libraries. However, at the time of this writing (September 2018), they all required the programmer either to learn a new syntax (PyContracts) or to write redundant condition descriptions (*e.g.*, contracts, covenant, deal, dpcontracts, pyadbc and pcd).

This library was strongly inspired by them, but we go two steps further.

First, our violation message on contract breach are much more informative. The message includes the source code of the contract condition as well as variable values at the time of the breach. This promotes don't-repeat-yourself principle (DRY) and spare the programmer the tedious task of repeating the message that was already written in code.

Second, icontract allows inheritance of the contracts and supports weakening of the preconditions as well as strengthening of the postconditions and invariants. Notably, weakening and strengthening of the contracts is a feature indispensable for modeling many non-trivial class hierarchies. Please see Section *Inheritance*. To the best of our knowledge, there is currently no other Python library that supports inheritance of the contracts in a correct way.

In the long run, we hope that design-by-contract will be adopted and integrated in the language. Consider this library a work-around till that happens. You might be also interested in the archived discussion on how to bring design-by-contract into Python language on python-ideas mailing list.

CHAPTER

TWO

USAGE

2.1 Preconditions and Postconditions

icontract provides two function decorators, *require* and *ensure* for pre-conditions and post-conditions, respectively. Additionally, it provides a class decorator, *invariant*, to establish class invariants.

The `condition` argument specifies the contract and is usually written in lambda notation. In post-conditions, condition function receives a reserved parameter `result` corresponding to the result of the function. The condition can take as input a subset of arguments required by the wrapped function. This allows for very succinct conditions.

You can provide an optional description by passing in `description` argument.

Whenever a violation occurs, *ViolationError* is raised. Its message includes:

- the human-readable representation of the condition,
- description (if supplied) and
- representation of all the values.

The representation of the values is obtained by re-executing the condition function programmatically by traversing its abstract syntax tree and filling the tree leaves with values held in the function frame. Mind that this re-execution will also re-execute all the functions. Therefore you need to make sure that all the function calls involved in the condition functions do not have any side effects.

If you want to customize the error, see Section *Custom Errors*.

```
>>> import icontract

>>> @icontract.require(lambda x: x > 3)
... def some_func(x: int, y: int = 5)->None:
...     pass
...

>>> some_func(x=5)

# Pre-condition violation
>>> some_func(x=1)
Traceback (most recent call last):
  ...
icontract.errors.ViolationError: File <doctest usage.rst[1]>, line 1 in <module>:
x > 3:
x was 1
y was 5
```

(continues on next page)

(continued from previous page)

```
# Pre-condition violation with a description
>>> @icontract.require(lambda x: x > 3, "x must not be small")
... def some_func(x: int, y: int = 5) -> None:
...     pass
...
>>> some_func(x=1)
Traceback (most recent call last):
  ...
icontract.errors.ViolationError: File <doctest usage.rst[4]>, line 1 in <module>:
x must not be small: x > 3:
x was 1
y was 5

# Pre-condition violation with more complex values
>>> class B:
...     def __init__(self) -> None:
...         self.x = 7
...
...     def y(self) -> int:
...         return 2
...
...     def __repr__(self) -> str:
...         return "an instance of B"
...
>>> class A:
...     def __init__(self)->None:
...         self.b = B()
...
...     def __repr__(self) -> str:
...         return "an instance of A"
...
>>> SOME_GLOBAL_VAR = 13
>>> @icontract.require(lambda a: a.b.x + a.b.y() > SOME_GLOBAL_VAR)
... def some_func(a: A) -> None:
...     pass
...
>>> an_a = A()
>>> some_func(an_a)
Traceback (most recent call last):
  ...
icontract.errors.ViolationError: File <doctest usage.rst[9]>, line 1 in <module>:
a.b.x + a.b.y() > SOME_GLOBAL_VAR:
SOME_GLOBAL_VAR was 13
a was an instance of A
a.b was an instance of B
a.b.x was 7
a.b.y() was 2

# Post-condition
>>> @icontract.ensure(lambda result, x: result > x)
... def some_func(x: int, y: int = 5) -> int:
```

(continues on next page)

(continued from previous page)

```
...     return x - y
...
>>> some_func(x=10)
Traceback (most recent call last):
  ...
icontract.errors.ViolationError: File <doctest usage.rst[12]>, line 1 in <module>:
result > x:
result was 5
x was 10
y was 5
```

2.2 Invariants

Invariants are special contracts associated with an instance of a class. An invariant should hold *after* initialization and *before* and *after* a call to any public instance method. The invariants are the pivotal element of design-by-contract: they allow you to formally define properties of a data structures that you know will be maintained throughout the life time of *every* instance.

We consider the following methods to be "public":

- All methods not prefixed with _
- All magic methods (prefix __ and suffix __)

Class methods (marked with `@classmethod` or special dunders such as `__new__`) can not observe the invariant since they are not associated with an instance of the class.

We exempt `__getattribute__`, `__setattr__` and `__delattr__` methods from observing the invariant since these functions alter the state of the instance and thus can not be considered "public".

We also exempt `__repr__` method to prevent endless loops when generating error messages.

The icontract invariants are implemented as class decorators.

The following examples show various cases when an invariant is breached.

After the initialization:

```
>>> @icontract.invariant(lambda self: self.x > 0)
... class SomeClass:
...     def __init__(self) -> None:
...         self.x = -1
...
...     def __repr__(self) -> str:
...         return "an instance of SomeClass"
...
>>> some_instance = SomeClass()
Traceback (most recent call last):
 ...
icontract.errors.ViolationError: File <doctest usage.rst[14]>, line 1 in <module>:
self.x > 0:
self was an instance of SomeClass
self.x was -1
```

Before the invocation of a public method:

```
>>> @icontract.invariant(lambda self: self.x > 0)
... class SomeClass:
...     def __init__(self) -> None:
...         self.x = 100
...
...     def some_method(self) -> None:
...         self.x = 10
...
...     def __repr__(self) -> str:
...         return "an instance of SomeClass"
...
>>> some_instance = SomeClass()
>>> some_instance.x = -1
>>> some_instance.some_method()
Traceback (most recent call last):
 ...
icontract.errors.ViolationError: File <doctest usage.rst[16]>, line 1 in <module>:
self.x > 0:
self was an instance of SomeClass
self.x was -1
```

After the invocation of a public method:

```
>>> @icontract.invariant(lambda self: self.x > 0)
... class SomeClass:
...     def __init__(self) -> None:
...         self.x = 100
...
...     def some_method(self) -> None:
...         self.x = -1
...
...     def __repr__(self) -> str:
...         return "an instance of SomeClass"
...
>>> some_instance = SomeClass()
>>> some_instance.some_method()
Traceback (most recent call last):
 ...
icontract.errors.ViolationError: File <doctest usage.rst[20]>, line 1 in <module>:
self.x > 0:
self was an instance of SomeClass
self.x was -1
```

After the invocation of a magic method:

```
>>> @icontract.invariant(lambda self: self.x > 0)
... class SomeClass:
...     def __init__(self) -> None:
...         self.x = 100
...
...     def __call__(self) -> None:
...         self.x = -1
...
```

(continues on next page)

(continued from previous page)

```
...     def __repr__(self) -> str:
...         return "an instance of SomeClass"
...
>>> some_instance = SomeClass()
>>> some_instance()
Traceback (most recent call last):
 ...
icontract.errors.ViolationError: File <doctest usage.rst[23]>, line 1 in <module>:
self.x > 0:
self was an instance of SomeClass
self.x was -1
```

2.3 Snapshots (a.k.a "old" argument values)

Usual postconditions can not verify the state transitions of the function's argument values. For example, it is impossible to verify in a postcondition that the list supplied as an argument was appended an element since the postcondition only sees the argument value as-is after the function invocation.

In order to verify the state transitions, the postcondition needs the "old" state of the argument values (*i.e.* prior to the invocation of the function) as well as the current values (after the invocation). *snapshot* decorator instructs the checker to take snapshots of the argument values before the function call which are then supplied as `OLD` argument to the postcondition function.

snapshot takes a capture function which accepts none, one or more arguments of the function. You set the name of the property in `OLD` as `name` argument to *snapshot*. If there is a single argument passed to the capture function, the name of the `OLD` property can be omitted and equals the name of the argument.

Here is an example that uses snapshots to check that a value was appended to the list:

```
>>> import icontract
>>> from typing import List

>>> @icontract.snapshot(lambda lst: lst[:])
... @icontract.ensure(lambda OLD, lst, value: lst == OLD.lst + [value])
... def some_func(lst: List[int], value: int) -> None:
...     lst.append(value)
...     lst.append(1984)  # bug

>>> some_func(lst=[1, 2], value=3)
Traceback (most recent call last):
    ...
icontract.errors.ViolationError: File <doctest usage.rst[28]>, line 2 in <module>:
lst == OLD.lst + [value]:
OLD was a bunch of OLD values
OLD.lst was [1, 2]
lst was [1, 2, 3, 1984]
result was None
value was 3
```

The following example shows how you can name the snapshot:

```
>>> import icontract
>>> from typing import List

>>> @icontract.snapshot(lambda lst: len(lst), name="len_lst")
... @icontract.ensure(lambda OLD, lst, value: len(lst) == OLD.len_lst + 1)
... def some_func(lst: List[int], value: int) -> None:
...     lst.append(value)
...     lst.append(1984)  # bug

>>> some_func(lst=[1, 2], value=3)
Traceback (most recent call last):
    ...
icontract.errors.ViolationError: File <doctest usage.rst[32]>, line 2 in <module>:
len(lst) == OLD.len_lst + 1:
OLD was a bunch of OLD values
OLD.len_lst was 2
len(lst) was 4
lst was [1, 2, 3, 1984]
result was None
value was 3
```

The next code snippet shows how you can combine multiple arguments of a function to be captured in a single snapshot:

```
>>> import icontract
>>> from typing import List

>>> @icontract.snapshot(
...     lambda lst_a, lst_b: set(lst_a).union(lst_b), name="union")
... @icontract.ensure(
...     lambda OLD, lst_a, lst_b: set(lst_a).union(lst_b) == OLD.union)
... def some_func(lst_a: List[int], lst_b: List[int]) -> None:
...     lst_a.append(1984)  # bug

>>> some_func(lst_a=[1, 2], lst_b=[3, 4])
Traceback (most recent call last):
    ...
icontract.errors.ViolationError: File <doctest usage.rst[36]>, line ... in <module>:
set(lst_a).union(lst_b) == OLD.union:
OLD was a bunch of OLD values
OLD.union was {1, 2, 3, 4}
lst_a was [1, 2, 1984]
lst_b was [3, 4]
result was None
set(lst_a) was {1, 2, 1984}
set(lst_a).union(lst_b) was {1, 2, 3, 4, 1984}
```

2.4 Inheritance

To inherit the contracts of the parent class, the child class needs to either inherit from *DBC* or have a meta class set to *icontract.DBCMeta*.

When no contracts are specified in the child class, all contracts are inherited from the parent class as-are.

When the child class introduces additional preconditions or postconditions and invariants, these contracts are *strengthened* or *weakened*, respectively. *icontract.DBCMeta* allows you to specify the contracts not only on the concrete classes, but also on abstract classes.

Strengthening. If you specify additional invariants in the child class then the child class will need to satisfy all the invariants of its parent class as well as its own additional invariants. Analogously, if you specify additional postconditions to a function of the class, that function will need to satisfy both its own postconditions and the postconditions of the original parent function that it overrides.

Weakining. Adding preconditions to a function in the child class weakens the preconditions. The caller needs to provide either arguments that satisfy the preconditions associated with the function of the parent class *or* arguments that satisfy the preconditions of the function of the child class.

Preconditions and Postconditions of __init__. Mind that `__init__` method is a special case. Since the constructor is exempt from polymorphism, preconditions and postconditions of base classes are *not* inherited for the `__init__` method. Only the preconditions and postconditions specified for the `__init__` method of the concrete class apply.

Abstract Classes. Since Python 3 does not allow multiple meta classes, *DBCMeta* inherits from abc.ABCMeta to allow combining contracts with abstract base classes.

Snapshots. Snapshots are inherited from the base classes for computational efficiency. You can use snapshots from the base classes as if they were defined in the concrete class.

The following example shows an abstract parent class and a child class that inherits and strengthens parent's contracts:

```
>>> import abc
>>> import icontract

>>> @icontract.invariant(lambda self: self.x > 0)
... class A(icontract.DBC):
...     def __init__(self) -> None:
...         self.x = 10
...
...     @abc.abstractmethod
...     @icontract.ensure(lambda y, result: result < y)
...     def func(self, y: int) -> int:
...         pass
...
...     def __repr__(self) -> str:
...         return "an instance of A"

>>> @icontract.invariant(lambda self: self.x < 100)
... class B(A):
...     def func(self, y: int) -> int:
...         # Break intentionally the postcondition
...         # for an illustration
...         return y + 1
...
...     def break_parent_invariant(self):
...         self.x = -1
```

(continues on next page)

(continued from previous page)

```
...
...     def break_my_invariant(self):
...         self.x = 101
...
...     def __repr__(self) -> str:
...         return "an instance of B"

# Break the parent's postcondition
>>> some_b = B()
>>> some_b.func(y=0)
Traceback (most recent call last):
    ...
icontract.errors.ViolationError: File <doctest usage.rst[40]>, line 7 in A:
result < y:
result was 1
self was an instance of B
y was 0

# Break the parent's invariant
>>> another_b = B()
>>> another_b.break_parent_invariant()
Traceback (most recent call last):
    ...
icontract.errors.ViolationError: File <doctest usage.rst[40]>, line 1 in <module>:
self.x > 0:
self was an instance of B
self.x was -1

# Break the child's invariant
>>> yet_another_b = B()
>>> yet_another_b.break_my_invariant()
Traceback (most recent call last):
    ...
icontract.errors.ViolationError: File <doctest usage.rst[41]>, line 1 in <module>:
self.x < 100:
self was an instance of B
self.x was 101
```

The following example shows how preconditions are weakened:

```
>>> class A(icontract.DBC):
...     @icontract.require(lambda x: x % 2 == 0)
...     def func(self, x: int) -> None:
...         pass

>>> class B(A):
...     @icontract.require(lambda x: x % 3 == 0)
...     def func(self, x: int) -> None:
...         pass
...
...     def __repr__(self) -> str:
...         return "an instance of B"
```

(continues on next page)

(continued from previous page)

```
>>> b = B()

# The precondition of the parent is satisfied.
>>> b.func(x=2)

# The precondition of the child is satisfied,
# while the precondition of the parent is not.
# This is OK since the precondition has been
# weakened.
>>> b.func(x=3)

# None of the preconditions have been satisfied.
>>> b.func(x=5)
Traceback (most recent call last):
    ...
icontract.errors.ViolationError: File <doctest usage.rst[49]>, line 2 in B:
x % 3 == 0:
self was an instance of B
x was 5
```

The example below illustrates how snapshots are inherited:

```
>>> class A(icontract.DBC):
...     @abc.abstractmethod
...     @icontract.snapshot(lambda lst: lst[:])
...     @icontract.ensure(lambda OLD, lst: len(lst) == len(OLD.lst) + 1)
...     def func(self, lst: List[int], value: int) -> None:
...         pass

>>> class B(A):
...     # The snapshot of OLD.lst has been defined in class A.
...     @icontract.ensure(lambda OLD, lst: lst == OLD.lst + [value])
...     def func(self, lst: List[int], value: int) -> None:
...         lst.append(value)
...         lst.append(1984)  # bug
...
...     def __repr__(self) -> str:
...         return "an instance of B"


>>> b = B()
>>> b.func(lst=[1, 2], value=3)
Traceback (most recent call last):
    ...
icontract.errors.ViolationError: File <doctest usage.rst[54]>, line 4 in A:
len(lst) == len(OLD.lst) + 1:
OLD was a bunch of OLD values
OLD.lst was [1, 2]
len(OLD.lst) was 2
len(lst) was 4
```

(continues on next page)

(continued from previous page)

```
lst was [1, 2, 3, 1984]
result was None
self was an instance of B
value was 3
```

2.5 Toggling Contracts

By default, the contract checks (including the snapshots) are always performed at run-time. To disable them, run the interpreter in optimized mode (`-O` or `-OO`, see Python command-line options).

If you want to override this behavior, you can supply the `enabled` argument to the contract:

```
>>> @icontract.require(lambda x: x > 10, enabled=False)
... def some_func(x: int) -> int:
...     return 123
...

# The pre-condition is breached, but the check was disabled:
>>> some_func(x=0)
123
```

Icontract provides a global variable `icontract.SLOW` to provide a unified way to mark a plethora of contracts in large code bases. `icontract.SLOW` reflects the environment variable `ICONTRACT_SLOW`.

While you may want to keep most contracts running both during the development and in the production, contracts marked with `icontract.SLOW` should run only during the development (since they are too sluggish to execute in a real application).

If you want to enable contracts marked with `icontract.SLOW`, set the environment variable `ICONTRACT_SLOW` to a non-empty string.

Here is some example code:

```
# some_module.py
@icontract.require(lambda x: x > 10, enabled=icontract.SLOW)
    def some_func(x: int) -> int:
        return 123

# in test_some_module.py
import unittest

class TestSomething(unittest.TestCase):
    def test_some_func(self) -> None:
        self.assertEqual(123, some_func(15))

if __name__ == '__main__':
    unittest.main()
```

Run this bash command to execute the unit test with slow contracts:

```
$ ICONTRACT_SLOW=true python test_some_module.py
```

2.6 Custom Errors

Icontract raises *ViolationError* by default. However, you can also instruct icontract to raise a different error by supplying `error` argument to the decorator.

The `error` argument can either be:

- **A callable that returns an exception.** The callable accepts the subset of arguments of the original function (including `result` and `OLD` for postconditions) or `self` in case of invariants, respectively, and returns an exception. The arguments to the condition function can freely differ from the arguments to the error function.

  The exception returned by the given callable is finally raised.

  If you specify the `error` argument as callable, the values will not be traced and the condition function will not be parsed. Hence, violation of contracts with `error` arguments as callables incur a much smaller computational overhead in case of violations compared to contracts with default violation messages for which we need to trace the argument values and parse the condition function.

- **A subclass of `BaseException`_.** The exception is constructed with the violation message and finally raised.
- **An instance of `BaseException`_.** The exception is raised as-is on contract violation.

Here is an example of the error given as a callable:

```
>>> @icontract.require(
...     lambda x: x > 0,
...     error=lambda x: ValueError('x must be positive, got: {}'.format(x)))
... def some_func(x: int) -> int:
...     return 123
...

# Custom Exception class
>>> some_func(x=0)
Traceback (most recent call last):
    ...
ValueError: x must be positive, got: 0
```

Here is an example of the error given as a subclass of BaseException:

```
>>> @icontract.require(lambda x: x > 0, error=ValueError)
... def some_func(x: int) -> int:
...     return 123
...

# Custom Exception class
>>> some_func(x=0)
Traceback (most recent call last):
    ...
ValueError: File <doctest usage.rst[62]>, line 1 in <module>:
x > 0: x was 0
```

Here is an example of the error given as an instance of a BaseException:

```
>>> @icontract.require(lambda x: x > 0, error=ValueError("x non-positive"))
... def some_func(x: int) -> int:
...     return 123
...
```

(continues on next page)

(continued from previous page)

```
# Custom Exception class
>>> some_func(x=0)
Traceback (most recent call last):
    ...
ValueError: x non-positive
```

Danger: Be careful when you write contracts with custom errors. This might lead the caller to (ab)use the contracts as a control flow mechanism.

In that case, the user will expect that the contract is *always* enabled and not only during debug or test. (For example, whenever you run Python interpreter with `-O` or `-OO`, `__debug__` will be `False`. If you left `enabled` argument to its default `__debug__`, the contract will *not* be verified in `-O` mode.)

2.7 Variable Positional and Keyword Arguments

Certain functions do not name their arguments explicitly, but operate on variable positional and/or keyword arguments supplied at the function call (*e.g.*, `def some_func(*args, **kwargs): ...`). Contract conditions thus need a mechanism to refer to these variable arguments. To that end, we introduced two special condition arguments, `_ARGS` and `_KWARGS`, that icontract will populate before evaluating the condition to capture the positional and keyword arguments, respectively, of the function call.

To avoid intricacies of Python's argument resolution at runtime, icontract simply captures *all* positional and keyword arguments in these two variables, regardless of whether the function defines them or not. However, we would recommend you to explicitly name arguments in your conditions and use `_ARGS` and `_KWARGS` only for the variable arguments for readability.

We present in the following a couple of valid contracts to demonstrate how to use these special arguments:

```
# The contract refers to the positional arguments of the *call*,
# though the decorated function does not handle
# variable positional arguments.
>>> @icontract.require(lambda _ARGS: _ARGS[0] > 0)
... def function_a(x: int) -> int:
...     return 123
>>> function_a(1)
123

# The contract refers to the keyword arguments of the *call*,
# though the decorated function does not handle variable keyword arguments.
>>> @icontract.require(lambda _KWARGS: _KWARGS["x"] > 0)
... def function_b(x: int) -> int:
...     return 123
>>> function_b(x=1)
123

# The contract refers both to the named argument and keyword arguments.
# The decorated function specifies an argument and handles
# variable keyword arguments at the same time.
>>> @icontract.require(lambda x, _KWARGS: x < _KWARGS["y"])
```

(continues on next page)

(continued from previous page)

```
... def function_c(x: int, **kwargs) -> int:
...     return 123
>>> function_c(1, y=3)
123

# The decorated functions accepts only variable keyboard arguments.
>>> @icontract.require(lambda _KWARGS: _KWARGS["x"] > 0)
... def function_d(**kwargs) -> int:
...     return 123
>>> function_d(x=1)
123

# The decorated functions accepts only variable keyboard arguments.
# The keyword arguments are given an uncommon name (``parameters`` instead
# of ``kwargs``).
>>> @icontract.require(lambda _KWARGS: _KWARGS["x"] > 0)
... def function_e(**parameters) -> int:
...     return 123
>>> function_e(x=1)
123
```

As a side note, we agree that the names picked for the placeholders are indeed a bit ugly. We decided against more aesthetic or ergonomic identifiers (such as `_` and `__` or `A` and `KW`) to avoid potential naming conflicts.

The underscore in front of the placeholders is meant to motivate a bit deeper understanding of the condition. For example, the reader needs to be aware that the logic for resolving the keyword arguments passed to the function is *different* in condition and that `_KWARGS` *does not* refer to arbitrary keyword arguments *passed to the condition*. Though this might be obvious for some readers, we are almost certain that `_ARGS` and `_KWARGS` will cause some confusion. We hope that a small hint like an underscore will eventually help the reading.

CHAPTER

THREE

CHECKING TYPES AT RUNTIME

Icontract focuses on logical contracts in the code. Theoretically, you could use icontract to check the types at runtime and condition the contracts using material implication:

```
@icontract.require(lambda x: not isinstance(x, int) or x > 0)
@icontract.require(lambda x: not isinstance(x, str) or x.startswith('x-'))
def some_func(x: Any) -> None
    ...
```

This is a good solution if your code lacks type annotations or if you do not know the type in advance.

However, if you already annotated the code with the type annotations, re-stating the types in the contracts breaks the DRY principle and makes the code unnecessarily hard to maintain and read:

```
@icontract.require(lambda x: isinstance(x, int))
def some_func(x: int) -> None
    ...
```

Elegant runtime type checks are out of icontract's scope. We would recommend you to use one of the available libraries specialized only on such checks such as typeguard.

The icontract's test suite explicitly includes tests to make sure that icontract and typeguard work well together and to enforce their interplay in the future.

CHAPTER

FOUR

ASYNC

Icontract supports both adding sync contracts to coroutine functions as well as enforcing *async conditions* (and capturing *async snapshots*).

You simply define your conditions as decorators of a coroutine function:

```
import icontract

@icontract.require(lambda x: x > 0)
@icontract.ensure(lambda x, result: x < result)
async def do_something(x: int) -> int:
    ...
```

4.1 Special Considerations

Async conditions. If you want to enforce async conditions, the function also needs to be defined as async:

```
import icontract

async def has_author(author_id: str) -> bool:
    ...

@icontract.ensure(has_author)
async def upsert_author(name: str) -> str:
    ...
```

It is not possible to add an async condition to a sync function. Doing so will raise a ValueError at runtime. The reason behind this limitation is that the wrapper around the function would need to be made async, which would break the code calling the original function and expecting it to be synchronous.

Invariants. As invariants need to wrap dunder methods, including `__init__`, their conditions *can not* be async, as most dunder methods need to be synchronous methods, and wrapping them with async code would break that constraint. You can, of course, use synchronous invariants on *async* method functions without problems.

No async lambda. Another practical limitation is that Python does not support async lambda (see this Python issue), so defining async conditions (and snapshots) is indeed tedious (see the *next section*). Please consider asking for async lambdas on python-ideas mailing list to give the issue some visibility. **Coroutine as condition result**. If the condition returns a coroutine, the coroutine will be awaited before it is evaluated for truthiness.

This means in practice that you can work around *no-async lambda limitation* applying coroutine functions on your condition arguments (which in turn makes the condition result in a coroutine).

For example:

```
async def some_condition(a: float, b: float) -> bool:
    ...

@icontract.require(lambda x: some_condition(a=x, b=x**2))
async def some_func(x: float) -> None:
    ...
```

A big fraction of contracts on sequences require an all operation to check that all the item of a sequence are True. Unfortunately, all does not automatically operate on a sequence of Awaitables, but the library asyncstdlib comes in very handy:

```
import asyncstdlib as a
```

Here is a practical example that uses asyncstdlib.map, asyncstdlib.all and asyncstdlib.await_each:

```
import asyncstdlib as a

async def has_author(identifier: str) -> bool:
    ...

async def has_category(category: str) -> bool:
    ...

@dataclasses.dataclass
class Book:
    identifier: str
    author: str

@icontract.require(lambda categories: a.map(has_category, categories))
@icontract.ensure(
    lambda result: a.all(a.await_each(has_author(book.author) for book in result)))
async def list_books(categories: List[str]) -> List[Book]:
    ...
```

Coroutines have side effects. If the condition of a contract returns a coroutine, the condition can not be re-computed upon the violation to produce an informative violation message. This means that you need to *specify an explicit error* which should be raised on contract violation.

For example:

```
async def some_condition() -> bool:
    ...

@icontract.require(
    lambda: some_condition(),
    error=lambda: icontract.ViolationError("Something went wrong."))
```

If you do not specify the error, and the condition returns a coroutine, the decorator will raise a ValueError at re-computation time.

CHAPTER

FIVE

RECIPES

Learning curve. In our experience, it does not take long for programmers to pick up and start applying design-by-contracts. Although only anecdotal, we observed that it takes junior programmers **one to two weeks** to get up to speed and start writing meaningful contracts in their code.

In general, writing contracts tends to be easy: while some conditions are really hard, most conditions are rather trivial. However, it takes a bit till it "clicks" so that writing contracts becomes a second nature when writing code. To help people with learning and training, we compiled a list of common recipes (*i.e.* patterns) that we often observed during the development.

Benefits of the recipes. In our experience, oftentimes it suffices if you just write down the obvious contracts. This will usually already substantially improve the correctness of your code and catch many bugs. To that end, recipes help you so that you do not have to spend so much mental energy on contracts, while reaping the benefits of the low-hanging fruits.

Suggestions. The following list of recipes is of course far from comprehensive. We collected the patterns based on our every-day programming, so they are intrinsically biased towards the areas we work on. If you developed contract patterns of your own that you do not see listed here, please feel free to suggest extensions! The easiest way for us to organize suggestions is if you create a new issue on our GitHub.

Corpus. We collected an extensive corpus of Python programs annotated with contracts, Python-by-contract corpus,. The corpus should serve both education purposes and as a testbed for the tools and libraries in the ecosystem.

We will use the snippets from the corpus as examples for the individual recipes. In the following, we will omit to put an explicit identifier of the corpus for brevity.

5.1 Serialize / Deserialize Pair

De/serialization data from different formats is a commonplace in many programs. Whenever you have a pair (a serialization and a de-serialization function), you should make it explicit in the contract how their inputs are connected.

For example, tokenizing a text and then converting the tokens back to text should give you the original input text.

From ethz_eprog_2019/exercise_11/problem_02.py#L159:

```
@ensure(
    lambda text, result:
    tokens_to_text(result) == text  # type: ignore
)
def tokenize(text: str) -> List[Token]:
    ...

@ensure(lambda tokens, result: tokens == tokenize(result))
```

(continues on next page)

(continued from previous page)

```
def tokens_to_text(tokens: Sequence[Token]) -> str:
    ...
```

This is a particularly nice scenario for auto-testing tools such as crosshair and icontract-hypothesis. Since the input is usually completely defined, the auto-testing tools can be readily used and often reveal relevant bugs in practice.

5.2 Encapsulation of Immutable Types

Certain contracts repeat throughout the code. If they operate on immutable data types, such as Sequence or int, you can encapsulate them into a new data type. Instead of using `__init__`, you use `__new__` for the construction and simply cast the input object to that new type *after* the pre-conditions were checked.

Consider `Lines`, a data structure representing an immutable sequence of strings which do not contain new-line characters.

From common.py#L18:

```
class Lines(DBC):
"""Represent a sequence of text lines."""

    # fmt: off
    @require(
        lambda lines:
        all('\n' not in line and '\r' not in line for line in lines)
    )
    # fmt: on
    def __new__(cls, lines: Sequence[str]) -> "Lines":
        return cast(Lines, lines)
```

Another good example is an identifier, a string complying to a certain pattern.

From ethz_eprog_2019/exercise_11/problem_02.py#L242:

```
IDENTIFIER_RE = re.compile(r"[a-zA-Z_][a-zA-Z0-9]*")


class Identifier(DBC, str):

    @require(lambda value: IDENTIFIER_RE.fullmatch(value))
    def __new__(cls, value: str) -> "Identifier":
        return cast(Identifier, value)
```

This pattern is very efficient — the underlying structure is left as-is and reaps all the benefits of efficient implementation — while it gives you additional static and runtime guarantees.

Note: If you use this pattern with icontract-hypothesis, you can not directly inherit from types such as `Sequence` as this will cause problems with the underlying Hypothesis library. Please see Hypothesis issue #2951.

For the time being (2021-07-01), the best approach is to annotate your data structure with all the methods that you actually need. See the class `Lines` at common.py#L18 for a full example.

5.3 Unique Elements in a Sequence

If you want to assert that the elements of a sequence are unique, you can use Pigeonhole Principle. Namely, if the elements in a collection are unique, the size of the corresponding set is equal to the size of the collection:

```
some_collection = list(...)
is_unique = len(set(some_collection)) == some_collection
```

From aoc2020/day_22_crab_combat.py#L40:

```
class Deck(DBC):
    """Represent a deck of cards."""

    @require(
        lambda cards:
        len(set(cards)) == len(cards),
        "Unique cards"
    )
    def __init__(self, cards: Sequence[int]) -> None:
        ...
```

5.4 Contracts on Elements of a Collection

We can use built-in functions all and any to model the contracts on the elements of a collection. Coupled with generator expressions, this gives very readable and elegant conditions.

From aoc2020/day_11_seating_system.py#L52:

```
import re

class Layout:
    """Represent a seat layout."""

    @require(
        lambda table:
        len(table) > 0
        and len(table[0]) > 0
        and all(
            len(row) == len(table[0])
            for row in table
        )
    )
    @require(
        lambda table:
        all(
            re.fullmatch(r"[L#.]", cell)
            for row in table
            for cell in row
        )
    )
    def __init__(self, table: List[List[str]]) -> None:
```

(continues on next page)

(continued from previous page)

```
        """Initialize with the given values."""
        ...
```

5.5 Elements of a Sequence Sorted

If we need to assert that the elements of a sequence are sorted, we can used the built-in function sorted to succinctly formulate the condition:

```
some_collection = list(...)
is_sorted = sorted(some_collection) == some_collection
```

From ethz_eprog_2019/exercise_03/problem_04.py#L38:

```
assert all("".join(sorted(key)) == key for key in TO_NUMBER)
```

5.6 Non-overlapping Sorted Ranges

Ranges (also called intervals) are usually defined with a start and an end. In many problems they are given in a sorted sequence where no overlap is expected. For example, if you have to model tags of a text, where the tags do not overlap.

Before we formulate the condition, let us first introduce a helper function to iterate over two consecutive elements in a sequence.

From common.py#L88:

```
def pairwise(iterable: Iterable[T]) -> Iterable[Tuple[T, T]]:
    """Iterate over ``(s0, s1, s2, ...)`` as ``((s0, s1), (s1, s2), ...)``."""
    previous = None  # type: Optional[T]
    for current in iterable:
        if previous is not None:
            yield previous, current

        previous = current
```

Assuming the ranges are sorted and non-overlapping, with inclusive start and exclusive end, the end of the `previous` range must equal the start of the `current` range.

From ethz_eprog_2019/exercise_12/problem_01.py#L88:

```
class Token(DBC):
    @require(lambda start, end: start < end)
    def __init__(self, ..., start: int, end: int, ...) -> None:
        ...
```

and from ethz_eprog_2019/exercise_12/problem_01.py#L147:

```
@ensure(
    lambda result:
    all(
        token1.end == token2.start
```

(continues on next page)

(continued from previous page)

```
        for token1, token2 in common.pairwise(result)
    ),
    "Tokens consecutive"
)
def tokenize(text: str) -> List[Token]:
    """Tokenize the given ``text``."""
    ...
```

If you want to allow "holes" between the ranges, just change the equality to the less comparison:

```
token1.end < token2.start
```

5.7 Material Conditional ("If ... then ...")

You can toggle contracts at load time of a module using `enabled` argument (see *Toggling Contracts*). This mechanism can not be used at runtime as we need toggling to be efficient and performed only once.

However, some contracts depend on the runtime values. For example, a condition on some value might apply only if the value is negative, or if a value is not `None`.

We can use the material conditional to formulate "if ... then ..." conditioning in the contracts. The implication, `A  B`, means that the logical statement `B` must hold if the logical statement `A` holds.

Unlike Eiffel Programming Language, Python does not provide the implication operator, but we can rewrite the material implication as: `¬A  B`. In Python, this is written as: `not A or B`.

From ethz_eprog_2019/exercise_05/problem_03.py#L217:

```
class Range:
    start: Final[float]
    end: Final[float]

    ...

@ensure(
    lambda ranges, value, result:
    not (value < ranges[0].start) or result == -1,
    "Value not covered in ranges => bin not found"
)
def bin_index(ranges: BinRanges, value: float) -> int:
    """Find the index of the bin range among ``ranges`` corresponding to ``value``."""
    ...
```

This pattern also nicely plays with sphinx-icontract which renders the material implication with the proper symbol . See the generated documentation of bin_index.

5.8 Compare against a Redundant Implementation

When you need to get a complex algorithm right, a common approach is to provide multiple redundant implementations. The hope is that the bugs will not replicate across the implementations. For example, the same task is given to different teams, or the same problem is tackled with different algorithms.

We observe often in practice that only two implementations are sufficient:

1) An optimized complex one, which has a high probability of bugs, and
2) A naïve inefficient one. This implementation takes substantially longer to run or does not scale with the large input at all. However, it is much easier to read and verify, and the probability of bugs is much smaller.

You assert then in code that the output of the optimized implementation matches the naïve one.

From ethz_eprog_2019/exercise_04/problem_01.py#L39, where we compute the Sieve of Eratosthenes:

```
@ensure(
    lambda result:
    all(
        naive_is_prime(number)
        for number in result
    )
)
def sieve(limit: int) -> List[int]:
    """
    Apply the Sieve of Eratosthenes on the numbers up to ``limit``.
    :return: list of prime numbers till ``limit``
    """
    ...
```

Usually you toggle this contract using `enabled` argument at load time so that it is only applied in the testing environments where you know that the input is small enough for the naïve implementation (see *Toggling Contracts*):

```
IN_TESTING_ENVIRONMENT = ...

@ensure(
    lambda result:
    all(
        naive_is_prime(number)
        for number in result
    ),
    enabled=IN_TESTING_ENVIRONMENT
)
```

Alternatively, you can use material conditional to limit the contract on small inputs at runtime (see the recipe *Material Conditional ("If ... then ...")*):

```
IN_TESTING_ENVIRONMENT = ...

@ensure(
    lambda result:
    not (max(result) < 100 and len(result) <= 10)
    or all(
        naive_is_prime(number)
        for number in result
```

(continues on next page)

(continued from previous page)

```
    )
)
```

5.9 Exclusive Or ("Either ... or ...")

Python already provides an exclusive or operator (^), so we can directly use it to model exclusive properties in the contracts. For example, a function returns either a valid result *or* an error message (but not both). Another example is if a function expects either one or the other argument to be specified (but, again, not both).

From ethz_eprog_2019/exercise_08/problem_05.py#L59:

```
@ensure(
    lambda pos, result:
    all(
        (next_pos.x == pos.x and next_pos.y != pos.y)
        ^ (next_pos.x != pos.x and next_pos.y == pos.y)
        for next_pos in result
    ),
    "Next is either in x- or in y-direction"
)
def list_next_positions(pos: Position) -> Sequence[Position]:
    """List all the possible next positions based on the current position ``pos``."""
    ...
```

5.10 Intermediate Variables

Long and complex contracts can become unwieldy, especially if certain expressions are re-used throughout the condition. Python's Assignment Expressions (a.k.a. walrus operator, :=) is particularly useful in such situations as it allows you to introduce a bit of "procedural" programming into the generally declarative conditions.

From ethz_eprog_2019/exercise_02/problem_03.py#L53:

```
@ensure(
    lambda result:
    all(
        (
                center := len(line) // 2,
                line[:center] == line[center + 1:][::-1]
                if len(line) % 2 == 1
                else line[:center] == line[center:][::-1]
        )[1]
        for line in result
    ),
    "Horizontal symmetry"
)
def draw(width: int, height: int) -> Lines:
    """Draw the pattern of the size ``width`` x ``height`` and return the text lines."""
    ...
```

Note: The walrus operator assigns the computed value to a variable *and* evaluates the value back. Hence we use a tuple to make the code "procedural" and re-use the assigned variable in the second element of the tuple.

Note: The introduction of the walrus operator (PEP 572) did spark some controversy and even lead to resigning of Guido van Rossum as Benevolent Dictator For Life (see this article about PEP 572 on lwn.net). With that said, you have to be careful not to abuse the feature.

For example, if the condition becomes *too* complex, it may be worth considering refactoring the code into a separate function (and testing it independently as well). Putting contract condition in a separate function has a disadvantage, though, that you do not get as informative violation messages (see Section *Usage*) and they hinder the tools like icontract-hypothesis.

CHAPTER

SIX

IMPLEMENTATION DETAILS

6.1 Decorator Stack

The precondition and postcondition decorators have to be stacked together to allow for inheritance. Hence, when multiple precondition and postcondition decorators are given, the function is actually decorated only once with a precondition/postcondition checker while the contracts are stacked to the checker's `__preconditions__` and `__postconditions__` attribute, respectively. The checker functions iterates through these two attributes to verify the contracts at run-time.

All the decorators in the function's decorator stack are expected to call functools.update_wrapper. Notably, we use `__wrapped__` attribute to iterate through the decorator stack and find the checker function which is set with functools.update_wrapper. Mind that this implies that preconditions and postconditions are verified at the inner-most decorator and *not* when outer preconditions and postconditions are defined.

Consider the following example:

```
@some_custom_decorator
@icontract.require(lambda x: x > 0)
@another_custom_decorator
@icontract.require(lambda x, y: y < x)
def some_func(x: int, y: int) -> None:
  # ...
```

The checker function will verify the two preconditions after both `some_custom_decorator` and `another_custom_decorator` have been applied, whily you would expect that the outer precondition (`x > 0`) is verified immediately after `some_custom_decorator` is applied.

To prevent bugs due to unexpected behavior, we recommend to always group preconditions and postconditions together.

6.2 Decoration with Invariants

Since invariants are handled by a class decorator (in contrast to function decorators that handle preconditions and postconditions), they do not need to be stacked. The first invariant decorator wraps each public method of a class with a checker function. The invariants are added to the class attribute `__invariants__`. At run-time, the checker function iterates through the `__invariants__` attribute when it needs to actually verify the invariants.

Mind that we still expect each class decorator that decorates the class functions to use functools.update_wrapper in order to be able to iterate through decorator stacks of the individual functions.

6.3 Recursion in Contracts

In certain cases functions depend on each other through contracts. Consider the following snippet:

```
@icontract.require(lambda: another_func())
def some_func() -> bool:
    ...

@icontract.require(lambda: some_func())
def another_func() -> bool:
    ...

some_func()
```

Naively evaluating such preconditions and postconditions would result in endless recursions. Therefore, icontract suspends any further contract checking for a function when re-entering it for the second time while checking its contracts.

Invariants depending on the instance methods would analogously result in endless recursions. The following snippet gives an example of such an invariant:

```
@icontract.invariant(lambda self: self.some_func())
class SomeClass(icontract.DBC):
    def __init__(self) -> None:
        ...

    def some_func(self) -> bool:
        ...
```

To avoid endless recursion icontract suspends further invariant checks while checking an invariant. The dunder `__dbc_invariant_check_is_in_progress__` is set on the instance for a diode effect as soon as invariant check is in progress and removed once the invariants checking finished. As long as the dunder `__dbc_invariant_check_is_in_progress__` is present, the wrappers that check invariants simply return the result of the function.

Invariant checks also need to be disabled during the construction since calling member functions would trigger invariant checks which, on their hand, might check on yet-to-be-defined instance attributes. See the following snippet:

```
@icontract.invariant(lambda self: self.some_attribute > 0)
class SomeClass(icontract.DBC):
    def __init__(self) -> None:
        self.some_attribute = self.some_func()

    def some_func(self) -> int:
        return 1984
```

CHAPTER

SEVEN

KNOWN ISSUES

7.1 Integration with `help()`

We wanted to include the contracts in the output of `help()`. Unfortunately, `help()` renders the `__doc__` of the class and not of the instance. For functions, this is the class "function" which you can not inherit from. See this discussion on python-ideas for more details.

7.2 Defining contracts outside of decorators

We need to inspect the source code of the condition and error lambdas to generate the violation message and infer the error type in the documentation, respectively. `inspect.getsource(.)` is broken on lambdas defined in decorators in Python 3.5.2+ (see this bug report). We circumvented this bug by using `inspect.findsource(.)`, `inspect.getsourcefile(.)` and examining the local source code of the lambda by searching for other decorators above and other decorators and a function or class definition below. The decorator code is parsed and then we match the condition and error arguments in the AST of the decorator. This is brittle as it prevents us from having partial definitions of contract functions or from sharing the contracts among functions.

Here is a short code snippet to demonstrate where the current implementation fails:

```
>>> import icontract

>>> require_x_positive = icontract.require(lambda x: x > 0)

>>> @require_x_positive
... def some_func(x: int) -> None:
...     pass

>>> some_func(x=0)
Traceback (most recent call last):
    ...
SyntaxError: Decorator corresponding to the line 1 could not be found in file <doctest␣
↪known_issues.rst[1]>: 'require_x_positive = icontract.require(lambda x: x > 0)\n'
```

However, we haven't faced a situation in the code base where we would do something like the above, so we are unsure whether this is a big issue. As long as decorators are directly applied to functions and classes, everything worked fine on our code base.

7.3 *args and **kwargs

Since handling variable number of positional and/or keyword arguments requires complex logic and entails many edge cases (in particular in relation to how the arguments from the actual call are resolved and passed to the contract), we did not implement it. These special cases also impose changes that need to propagate to rendering the violation messages and related tools such as pyicontract-lint and sphinx-icontract. This is a substantial effort and needs to be prioritized accordingly.

Before we spend a large amount of time on this feature, please give us a signal through the issue 147 and describe your concrete use case and its relevance. If there is enough feedback from the users, we will of course consider implementing it.

7.4 dataclasses

When you define contracts for dataclasses, make sure you define the contracts *after* decorating the class with `@dataclass` decorator:

```
>>> import icontract
>>> import dataclasses

>>> @icontract.invariant(lambda self: self.x > 0)
... @dataclasses.dataclass
... class Foo:
...     x: int = dataclasses.field(default=42)
```

This is necessary as we can not re-decorate the methods that `dataclass` decorator inserts.

CHAPTER

EIGHT

BENCHMARKS

We run benchmarks against *deal* and *dpcontracts* libraries as part of our continuous integration.

The bodies of the constructors and functions were intentionally left simple so that you can better estimate **overhead** of the contracts in absolute terms rather than relative. This means that the code without contracts will run extremely fast (nanoseconds) in the benchmarks which might make the contracts seem sluggish. However, the methods in the real world usually run in the order of microseconds and milliseconds, not nanoseconds. As long as the overhead of the contract is in the order of microseconds, it is often practically acceptable.

The following scripts were run:

- benchmarks/against_others/compare_invariant.py
- benchmarks/against_others/compare_precondition.py
- benchmarks/against_others/compare_postcondition.py

The benchmarks were executed on Intel(R) Xeon(R) E-2276M CPU @ 2.80GHz. We used Python 3.9.9, icontract 2.6.1, deal 4.23.3 and dpcontracts 0.6.0.

The following tables summarize the results.

Benchmarking invariant at __init__:

Case	Total time	Time per run	Relative time per run
ClassWithIcontract	1.45 s	1.45 s	100%
ClassWithDpcontracts	0.48 s	0.48 s	33%
ClassWithDeal	1.73 s	1.73 s	119%
ClassWithInlineContract	0.28 s	0.28 s	19%

Benchmarking invariant at a function:

Case	Total time	Time per run	Relative time per run
ClassWithIcontract	2.04 s	2.04 s	100%
ClassWithDpcontracts	0.49 s	0.49 s	24%
ClassWithDeal	4.67 s	4.67 s	230%
ClassWithInlineContract	0.23 s	0.23 s	11%

Benchmarking precondition:

Case	Total time	Time per run	Relative time per run
function_with_icontract	0.04 s	3.91 s	100%
function_with_dpcontracts	0.54 s	53.92 s	1377%
function_with_deal	0.04 s	4.16 s	106%
function_with_inline_contract	0.00 s	0.15 s	4%

Benchmarking postcondition:

Case	Total time	Time per run	Relative time per run
function_with_icontract	0.04 s	4.39 s	100%
function_with_dpcontracts	0.53 s	52.51 s	1197%
function_with_deal_post	0.01 s	1.16 s	26%
function_with_deal_ensure	0.01 s	1.04 s	24%
function_with_inline_contract	0.00 s	0.15 s	3%

Note that neither the *dpcontracts* nor the *deal* library support recursion and inheritance of the contracts. This allows them to use faster enforcement mechanisms and thus gain a speed-up.

We also ran a much more extensive battery of benchmarks on icontract 2.0.7. Unfortunately, it would cost us too much effort to integrate the results in the continuous integration. The report is available at: benchmarks/benchmark_2.0.7.rst.

The scripts are available at: benchmarks/import_cost/ and benchmarks/runtime_cost/. Please re-run the scripts manually to obtain the results with the latest icontract version.

CHAPTER

NINE

API

9.1 require

class icontract.**require**

Decorate a function with a precondition.

The arguments of the precondition are expected to be a subset of the arguments of the wrapped function.

__init__(*condition: ~typing.Callable[[...], ~typing.Any], description: ~typing.Optional[str] = None, a_repr: ~reprlib.Repr = <reprlib.Repr object>, enabled: bool = True, error: ~typing.Optional[~typing.Union[~typing.Callable[[...], ~icontract._globals.ExceptionT], ~typing.Type[~icontract._globals.ExceptionT], BaseException]] = None*) → None

Initialize.

Parameters

- **condition** – precondition predicate

  If the condition returns a coroutine, you must specify the *error* as coroutines have side effects and can not be recomputed.

- **description** – textual description of the precondition
- **a_repr** – representation instance that defines how the values are represented
- **enabled** – The decorator is applied only if this argument is set.

  Otherwise, the condition check is disabled and there is no run-time overhead.

  The default is to always check the condition unless the interpreter runs in optimized mode (`-O` or `-OO`).

- **error** – The error is expected to denote either:
  - A callable. `error` is expected to accept a subset of function arguments and return an exception. The `error` is called on contract violation and the resulting exception is raised.
  - A subclass of `BaseException` which is instantiated with the violation message and raised on contract violation.
  - An instance of `BaseException` that will be raised with the traceback on contract violation.

__call__(*func: CallableT*) → CallableT

Add the precondition to the list of preconditions of the function `func`.

The function `func` is decorated with a contract checker if there is no contract checker in the decorator stack.

Parameters
func – function to be wrapped

Returns
contract checker around `func` if no contract checker on the decorator stack, or `func` otherwise

9.2 snapshot

class icontract.**snapshot**

Decorate a function with a snapshot of argument values captured *prior* to the function invocation.

A snapshot is defined by a capture function (usually a lambda) that accepts one or more arguments of the function. If the name of the snapshot is not given, the capture function must have a single argument and the name is equal to the name of that single argument.

The captured values are supplied to postconditions with the OLD argument of the condition and error function. Snapshots are inherited from the base classes and must not have conflicting names in the class hierarchy.

__init__(*capture: Callable[[...], Any]*, *name: Optional[str] = None*, *enabled: bool = True*) → None

Initialize.

Parameters

- **capture** – function to capture the snapshot accepting a one or more arguments of the original function *prior* to the execution
- **name** – name of the snapshot; if omitted, the name corresponds to the name of the input argument
- **enabled** – The decorator is applied only if `enabled` is set.

  Otherwise, the snapshot is disabled and there is no run-time overhead.

  The default is to always capture the snapshot unless the interpreter runs in optimized mode (`-O` or `-OO`).

__call__(*func: CallableT*) → CallableT

Add the snapshot to the list of snapshots of the function `func`.

The function `func` is expected to be decorated with at least one postcondition before the snapshot.

Parameters
func – function whose arguments we need to snapshot

Returns
`func` as given in the input

9.3 ensure

class icontract.**ensure**

Decorate a function with a postcondition.

The arguments of the postcondition are expected to be a subset of the arguments of the wrapped function. Additionally, the argument "result" is reserved for the result of the wrapped function. The wrapped function must not have "result" among its arguments.

__init__(*condition: ~typing.Callable[[...], ~typing.Any], description: ~typing.Optional[str] = None, a_repr: ~reprlib.Repr = <reprlib.Repr object>, enabled: bool = True, error: ~typing.Optional[~typing.Union[~typing.Callable[[...], ~icontract._globals.ExceptionT], ~typing.Type[~icontract._globals.ExceptionT], BaseException]] = None*) → None

Initialize.

Parameters

- **condition** – postcondition predicate.

  If the condition returns a coroutine, you must specify the *error* as coroutines have side effects and can not be recomputed.

- **description** – textual description of the postcondition
- **a_repr** – representation instance that defines how the values are represented
- **enabled** – The decorator is applied only if this argument is set.

  Otherwise, the condition check is disabled and there is no run-time overhead.

  The default is to always check the condition unless the interpreter runs in optimized mode (`-O` or `-OO`).

- **error** – The error is expected to denote either:
  - A callable. `error` is expected to accept a subset of function arguments and return an exception. The `error` is called on contract violation and the resulting exception is raised.
  - A subclass of `BaseException` which is instantiated with the violation message and raised on contract violation.
  - An instance of `BaseException` that will be raised with the traceback on contract violation.

__call__(*func: CallableT*) → CallableT

Add the postcondition to the list of postconditions of the function `func`.

The function `func` is decorated with a contract checker if there is no contract checker in the decorator stack.

Parameters
func – function to be wrapped

Returns
contract checker around `func` if no contract checker on the decorator stack, or `func` otherwise

9.4 invariant

class icontract.**invariant**

Represent a class decorator to establish the invariant on all the public methods.

Class method as well as "private" (prefix __) and "protected" methods (prefix _) may violate the invariant. Note that all magic methods (prefix __ and suffix __) are considered public and hence also need to establish the invariant. To avoid endless loops when generating the error message on an invariant breach, the method `__repr__` is deliberately exempt from observing the invariant.

The invariant is checked *before* and *after* the method invocation.

As invariants need to wrap dunder methods, including `__init__`, their conditions *can not* be async, as most dunder methods need to be synchronous methods, and wrapping them with async code would break that constraint.

__init__(*condition: ~typing.Callable[[...], ~typing.Any], description: ~typing.Optional[str] = None, a_repr: ~reprlib.Repr = <reprlib.Repr object>, enabled: bool = True, error: ~typing.Optional[~typing.Union[~typing.Callable[[...], ~icontract._globals.ExceptionT], ~typing.Type[~icontract._globals.ExceptionT], BaseException]] = None*) → None

Initialize a class decorator to establish the invariant on all the public methods.

Parameters

- **condition** – invariant predicate.

  The condition must not be a coroutine function as dunder functions (including `__init__`) of a class can not be async.

- **description** – textual description of the invariant
- **a_repr** – representation instance that defines how the values are represented
- **enabled** – The decorator is applied only if this argument is set.

  Otherwise, the condition check is disabled and there is no run-time overhead.

  The default is to always check the condition unless the interpreter runs in optimized mode (`-O` or `-OO`).

- **error** – The error is expected to denote either:
  - A callable. `error` is expected to accept a subset of function arguments and return an exception. The `error` is called on contract violation and the resulting exception is raised.
  - A subclass of `BaseException` which is instantiated with the violation message and raised on contract violation.
  - An instance of `BaseException` that will be raised with the traceback on contract violation.

Returns

__call__(*cls: ClassT*) → ClassT

Decorate each of the public methods with the invariant.

Go through the decorator stack of each function and search for a contract checker. If there is one, add the invariant to the checker's invariants. If there is no checker in the stack, wrap the function with a contract checker.

9.5 DBCMeta

class icontract.**DBCMeta**(*name*, *bases*, *namespace*, ***kwargs*)

Define a meta class that allows inheritance of the contracts.

The preconditions are weakened ("require else"), while postconditions ("ensure then") and invariants are strengthened according to the inheritance rules of the design-by-contract.

9.6 DBC

class icontract.**DBC**

Provide a standard way to create a class which can inherit the contracts.

9.7 ViolationError

class icontract.**ViolationError**

Indicate a violation of a contract.

CHAPTER

TEN

DEVELOPMENT

- Check out the repository.
- In the repository root, create the virtual environment:

```
python3 -m venv venv3
```

- Activate the virtual environment:

```
source venv3/bin/activate
```

- Install the development dependencies:

```
pip3 install -e .[dev]
```

- We use tox for testing and packaging the distribution. Run:

```
tox
```

- We also provide a set of pre-commit checks that lint and check code for formatting. Run them locally from an activated virtual environment with development dependencies:

```
./precommit.py
```

- The pre-commit script can also automatically format the code:

```
./precommit.py  --overwrite
```

10.1 Commit Message Style

Use the following guidelines for commit message.

- Past tense in the subject & body
- Max. 50 characters subject
- Max. 72 characters line length in the body (multiple lines are ok)
- Past tense in the body
- Have separate commits for the releases where the important changes are highlighted

See examples from past commits at https://github.com/Parquery/icontract/commits/master/

CHAPTER

ELEVEN

CHANGELOG

11.1 2.6.6

- Updated typeguard and deal to latest versions (#284)

  This change is needed so that distributions can successfully run the necessary tests with the development dependencies. Previously, the dependencies were outdated, and the old versions were already deprecated in distributions (notably, typegard and deal).

11.2 2.6.5

- Added Python 3.11 to the list of supported Pythons (#280)
- Fixed deal dependency marker (#279)

  This patch is important as we silently broke `setup.py`, which was tolerated by older versions of setuptools, but not any more by the newer ones. With this patch, icontract's `setup.py` is made valid again.

11.3 2.6.4

- Restored Python 3.6 support (#274)

  The support for Python 3.6 has been dropped in #257 as GitHub removed its support in the CI pipeline. With this patch, we restored the support of Python 3.6. Notably, we had to add the package `contextvars` conditioned on Python 3.6.

11.4 2.6.3

- Removed meta data files from setup.py (#262)
- Added support for python 3.11 (#260)
- Fixed in-progress set for async (#256)

11.5 2.6.2

- Added wheels to releases (#251)
- Fixed mypy error on missing `asttokens.ASTTokens` (#252)

11.6 2.6.1

- Excluded all tests from package (#240)

11.7 2.6.0

- Added support for Python 3.9 and 3.10 (#236)
- Added representation of subscripts (#237)

11.8 2.5.5

- Fixed representation of numpy arrays (#232)
- Removed tag for Python 3.5 (#231)

11.9 2.5.4

- Made type annotation for `invariant` decorator more specific (#227)

11.10 2.5.3

- Fixed reporting all arguments on violation (#219)
- Propagated placeholders in re-computation (#218)
- Fixed docstring for `collect_variable_lookup` (#217)

11.11 2.5.2

- Fixed handling of `self` when passed as kwarg (#213)
- Added reporting of all arguments on violation (#214)
- Added tracing of `all` on generator expressions (#215)

11.12 2.5.1

- Allowed `__new__` to tighten pre-conditions (#211)
- Fixed recomputation of calls in generator expr (#210)
- Added better reporting on recompute failure (#207)

11.13 2.5.0

- Encapsulated adding contracts for integrators (#202)
- Added support for error-as-instance (#201)
- Added support for coroutine (#197)
- Added support for async (#196)

11.14 2.4.1

- Removed automatic registration with Hypothesis and replaced it with a hook that downstream libraries such as icontract-hypothesis can use (#181)
- Refactored and added tests for integrators (#182)

11.15 2.4.0

- Integrated with icontract-hypothesis (#179)
- Refactored for icontract-hypothesis (#178)
- Added special arguments *_ARGS* and *_KWARGS* (#176)
- Tested with typeguard (#175)
- Tested with *dataclasses.dataclass* (#173)
- Added invariants to namedtuple (#172)
- Added support for recomputation of f-strings (#170)
- Exempted *__new__* from invariant checks (#168)
- Added support for named expressions in contracts (#166)

11.16 2.3.7

- Acted upon deprecation warning ins `ast` module when generating the violation error message.

11.17 2.3.6

- Denormalized icontract_meta so that icontract can be installed on readthedocs.

11.18 2.3.5

- Disabled invariant checks during the construction to avoid attribute errors on uninitialized attributes

11.19 2.3.4

- Added `icontract_meta` to `setup.py`
- Noted that contracts on `*args` and `**kwargs` are known issues

11.20 2.3.3

- Fixed performance regression due to state

11.21 2.3.2

- Fixed bug related to threading.local and mutables

11.22 2.3.1

- Fixed race conditions in endless recursion blockers

11.23 2.3.0

- Disabled recursion in the contracts
- Upgraded min version of asttokens to 2

11.24 2.2.0

- Made compatible with Python 3.8

11.25 2.1.0

- Made snapshot accept multiple arguments

11.26 2.0.7

- Fixed mypy complaints in clients due to import aliases
- Made compliant to mypy 0.750 –strict

11.27 2.0.6

- Added location to errors on calls with missing arguments

11.28 2.0.5

- Improved error message on unexpected arguments in a call
- Distinguished between optional and mandatory arguments in conditions. Default argument values in conditions are accepted instead of raising a misleading "missing argument" exception.
- Added a boolyness check to detect if the condition evaluation can be negated. If the condition evaluation lacks boolyness, a more informative exception is now raised. For example, this is important for all the code operating with numpy arrays where boolyness is not given.
- Added contract location to `require`, `ensure` and `snapshot`. This feature had been erroneously omitted in 2.0.4.

11.29 2.0.4

- Added contract location to the message of the violation error

11.30 2.0.3

- Fixed representation of numpy conditions
- Updated pylint to 2.3.1

11.31 2.0.2

- Specified `require` and `ensure` to use generics in order to fix typing erasure of the decorated functions

11.32 2.0.1

- Fixed forgotten renamings in the Readme left from icontract 1.x

11.33 2.0.0

- Removed `repr_args` argument to contracts since it is superseded by more versatile `error` argument
- Renamed contracts to follow naming used in other languages and libraries (`require`, `ensure` and `invariant`)
- Improved error messages on missing arguments in the call

11.34 1.7.2

- Demarcated decorator and lambda inspection in `_represent` submodule

11.35 1.7.1

- Refactored implementation and tests into smaller modules

11.36 1.7.0

- Added `snapshot` decorator to capture "old" values (prior to function invocation) for postconditions that verify state transitions

11.37 1.6.1

- Replaced `typing.Type` with `type` so that icontract works with Python 3.5.2

11.38 1.6.0

- Added `error` argument to the contracts

11.39 1.5.9

- Removed `ast_graph` module which was only used for debugging
- Prefixed internal modules with an underscore (`_represent` and `_recompute`)

11.40 1.5.8

- `recompute` propagates to children of generator expressions and comprehensions
- Optimized parsing of condition lambdas by considering only lines local to the decorator

11.41 1.5.7

- Exempted `__init__` from inheritance of preconditions and postconditions if defined in the concrete class.

11.42 1.5.6

- Contracts are observed and inherited with property getters, setters and deleters.
- Weakining of preconditions of a base function without any preconditions raises `TypeError`.
- `__getattribute__`, `__setattr__` ``and ``__delattr__` are exempted from invariants.
- Slot wrappers are properly handled.
- Fixed representation of conditions with attributes in generator expressions
- Added reference to sphinx-contract

11.43 1.5.5

- Added reference to `pyicontract-lint` in the README
- Made `inv` a class

11.44 1.5.4

- Added support for class and static methods

11.45 1.5.3

- Fixed different signatures of `DBCMeta` depending on Python version (<=3.5 and >3.5) due to differing signatures of `__new__` in `abc.ABCMeta`

11.46 1.5.2

- Removed dependency on `meta` package and replaced it with re-parsing the file containing the condition to represent the comprehensions

11.47 1.5.1

- Quoted ellipsis in `icontract._unwind_decorator_stack` to comply with a bug in Python 3.5.2 (see https://github.com/python/typing/issues/259)

11.48 1.5.0

- Added inheritance of contracts

11.49 1.4.1

- Contract's constructor immediately returns if the contract is disabled.

11.50 1.4.0

- Added invariants as *icontract.inv*

11.51 1.3.0

- Added `icontract.SLOW` to mark contracts which are slow and should only be enabled during development
- Added `enabled` flag to toggle contracts for development, production __etc.__

11.52 1.2.3

- Removed `version.txt` that caused problems with `setup.py`

11.53 1.2.2

- Fixed: the `result` is passed to the postcondition only if necessary

11.54 1.2.1

- Fixed a bug that fetched the unexpected frame when conditions were stacked
- Fixed a bug that prevented default function values propagating to the condition function

11.55 1.2.0

- Added reprlib.Repr as an additional parameter to customize representation

11.56 1.1.0

- Fixed unit tests to set actual and expected arguments correctly
- Made ViolationError an AssertionError
- Added representation of values by re-executing the abstract syntax tree of the function

11.57 1.0.3

- `pre` and `post` decorators use `functools.update_wrapper` to allow for doctests

11.58 1.0.2

- Moved icontract.py to a module directory
- Added py.typed to comply with mypy

11.59 1.0.1

- Fixed links in the README and setup.py

11.60 1.0.0

- Initial version

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

D

E

I

R

S

V