CodeVikng.Contract Documentation

Release 0.12.4

Dan Bullok

March 04, 2015

Contents

CHAPTER 1

Overview

This package provides simple but powerful support for contract programming. It includes support for preconditions, postconditions, invariants, and function signature checking. Decorators are used to specify preconditions, postconditions, and invariants. Function signatures are automatically extracted from argument and return type annotations. All contracts can easily be enabled or disabled. Disabled contracts add zero runtime overhead.

This package currently supports Python 3 only. There is no planned support for Python 2.

WARNING: Code and documentation are currently in alpha testing. Everything that has been impelemented is expected to work, with the following exceptions:

- **lazily-parsed argument and return value annotations.**

CHAPTER 2

Features

CodeViking Contracts supports the following contract programming features:

Precondition A condition that must be True before a function is called.

Postcondition A condition that must be True after a function has returned.

Invariant A property whose value must not change over the course of a function call. This is essentially a pre- and post- condition rolled into one.

Note: Our use of *invariant* is slightly different than the one traditionally used in the contract programming community. A codeviking .contracts invariant is a property that is unchanged by the execution of a function. To clarify:

typical invariant a condition that is always true throughout the life of a function call or class instance.

our invariant We adopt the definition used in mathematics. Given a property *p()* and a function *f(*args, **kwargs)*, let:

- *p_before = p()*
- *f(*args, **kwargs)*
- *p_after = p()*

We say a property is invariant if *p_before == p_after*.

Checker Ensures that a value satisfies arbitrary constraints. We support checking argument values passed to a function, and values returned from a function. Although these are two different conditions, they are specified using the same set of rules.

Preconditions, postconditions, and invariants are specified using function decorators. Checkers are automatically extracted from function annotations.

CHAPTER 3

Quick Start

3.1 Install

```
pip3 install codeviking.contract
```

3.2 Usage

```
from codeviking.contracts import contract

@check_sig
class Stack:

    @postcondition(is_empty)
    def __init__(self):
        ...

    def push(self, element: Elem):
        ...

    @precondition(lambda s: not s.is_empty)
    def pop(self) -> Elem:
        ...

    def length(self) -> int:
        ...
```

CHAPTER 4

Argument and Return Value Conditions

Validating argument and return values are crucial to creating a function contract. Conditions on function arguments and return values can be automatically enforced:

```
def f(x: float, y:float) -> Point2d:
    pass

def add(name: str,
        elem: lambda(e): hasattr(elem, 'e_type') and
                         e.isValid ) -> Point2d:
    pass

def scale(x: {int,float}) -> Option(List(Shape)):
    pass
```

There is no need to specify conditions for every argument:

```
def lookup(key, timeout: float):
    pass

def read(stream) -> List(Token):
    pass

def remove(key):
    pass
```

CHAPTER 5

Preconditions, Postconditions, and Invariants

Specifying other conditions is slightly more verbose

```
@precondition(lambda s: not s.is_empty):
def pop(self) -> Element:

@invariant(is_valid)
def compact(self):
@postcondition(is_empty)
def clear(self):
```

CHAPTER 6

Checkers Specification

Checkers are automatically extracted from function annotations:

```
def find(self, elem: (int, str), start_index: int = 0) -> Option(int):
    ...
```

In the above function:

- the checker *(int, str)* will be applied to argument *elem*
- the checker *int* will be applied to argument *start_index*
- the checker *Option(int)* will be applied to the return value of *find(...)*

There are many ways to specify checkers. In the following examples, *arg* is the argument or return value that is being validated, and *c, c0, c1, ...* are arbitrary checkers.

Table 6.1: Built-in Checkers

Annotation	Meaning	Example	Success Condition
type	Type Verification	`arg: MyClass`	*arg* is an instance of class *MyClass*
set literal	Disjunction (or)	`arg: {c0, c1, ..., ck}`	*arg* must satisfy at least one of the conditions *c0, c1, ..., ck*
list literal	Conjunction (and)	`arg: [c0, c1, ..., ck]`	*arg* must satisfy all of the conditions *c0, c1, ..., ck*
tuple	Element-wise Condition	`arg: (c0, c1, ..., ck)`	*arg* is a tuple of length *k* whose elements satisfy conditions *c0, c1, ..., ck*
str	Delayed Evaluation	`arg: "check_expr"`	The string "check_expr" is evaluated to produce the condition. This can be used to specify a type or other condition that has not yet been defined; it is similar to a forward declaration in languages like C++.
function	Functional Condition	`arg: func`	*func(arg)* is true
`Option`	Optional match	`arg: Option(c)`	condition *c* succeeds, or *arg is None*.
`Is`	Identical match	`arg: Is(obj)`	*arg is obj*
`List`	List	`arg: List(c)`	*arg* is a list, and all the elements in *arg* satisfy *c*.
`Dict`	Dictionary	`arg: Dict(c_key, c_value)`	*arg* is a dict, *c_key* succeeds for the keys of *arg*, and *c_value* succeeds for the values of'arg'.
`Any`	Universal Match	`arg: Any`	*arg* can be anything (condition is always satisfied)

NamedTuple - used to create namedtuples whose elements are subject to contracts.

6.1 Nested Conditions

Conditions may be nested. For example:

`arg: (int, {float, int}, List(List(str)))`

matches a tuple with the following properties:

- the first element must be of type int
- the second element must be of type int or type float
- the third element must be a list whose elements are lists of str.

`Option([f,{g,h}])`

where f, g, and h are functions, matches:

- None or

- arg for which `f(arg) == True`, and at least one of `g(arg)` and `h(arg)` are True

CHAPTER 7

Disabling Contracts

Contracts are enabled by default, but can be enabled or disabled as often as desired by setting `contracts.enabled` = TruelFalse. For example:

```
from codeviking.contracts import contracts, check_sig, precondition

# contracts are enabled by default, so f1 will be checked.
@check_sig
def f1(x: float, y: float) -> float:
    ...

contracts.enabled = False
# contracts are disabled until we turn them back on.
# f2 will not have its signature checked
def f2(x: float, y: float) -> float:
    ...

# contracts are still disabled
@check_sig
class A:
    def __init__(self, a:int, b:int):
        ...
    @precondition(lambda s: s.a != s.b)
    def skew(self, s: float):
        ...

contracts.enabled = True
# everything from here on will be checked (unless contracts are disabled
again).
```

Note that contracts can only be enabled/disabled at the time a function or class is defined. If contracts are disabled, they do not wrap the decorated class or function, so there is no runtime penalty.

It is technically possible to implement runtime contract enabling/disabling, though there will be a small run-time penalty in this case. If you are interested in this feature, please make a feature request on the project website.

CHAPTER 8

Errors

[ContractViolation] - This represents a violation of some part of a contract. The contract library allows subclasses of `ContractViolation` to pass up to the user so they can be notified of the code that caused the contract violation.

Subclasses:

- `SignatureViolation`
- `PreconditionViolation`
- `InvariantViolation`
- `PostconditionViolation`

`ContractError` - This is distinct from a `ContractViolation`. It does not indicate a failed contract. Rather, it indicates that some sort of problem was encountered while attempting to validate a contract condition. This is a serious error. It means that there is a bug in a contract, or in some code that a contract calls.

Subclasses:

- `SignatureError`
- `PreconditionError`
- `InvariantError`
- `PostconditionError`

CHAPTER 9

How does it all work?

All contract decorators that operate on the same function are combined into a single `ContractWrapper` object. Instead of nested wrappers (what you normally get when you use multiple decorators), we collect all of the contracts together and execute them in the proper sequence. This makes stack traces easier to follow, and is slightly more efficient. A `ContractWrapper` works as follows:

- Call all of the preconditions. If any of them return False,we raise a `PreconditionViolation`. If any of the preconditions raise an exception other than a ContractViolation we catch it and raise a PreconditionError.
- Call each of the invariant properties and store them. If any of them raise a ContractViolation, we ignore it. Any other exceptions are caught and we raise an InvariantError.
- Call the actual wrapped function and save the return value.
- Call each of the invariant properties once again. We handle any exceptions as before. If any of the invariant properties fail to match the value we saved before the function call, we raise an InvariantViolation exception.
- Call all of the postconditions. If any of them return False, we raise a PostconditionViolation. If any of the postconditions raise an exception other than a ContractViolation we catch it and raise a PostconditionError.
- We return the stored return value of the wrapped function.

Try to keep all of the contracts together. If you use any other decorators (like @property), put them at the top of your decorators.

CHAPTER 10

codeviking.contracts Package

10.1 Decorators

contracts
A Switch instance that controls all of the decorators in the package: precondition, postcondition, invariant, and check_sig

@check_sig
When applied to a function, enable signature checking by extracting argument and return type annotations. When applied to a class, enable checking for all annotated member functions within the class.

@invariant(*prop*)
Evaluate prop before and after the decorated function is called. If the values match, the invariant is satisfied.

@precondition(*cond*)
Evaluate cond before and decorated function is called. If cond is True, the precondition is satisfied.

@postcondition(*cond*)
Evaluate cond before and decorated function is called. If cond is True, the postcondition is satisfied.

10.2 Checkers

These are used to Annotate function arguments and return values.

Any
Matches any argument (always succeeds).

class Is(*other*)
Matches if `(argument is other)` is True

class Option(*other*)
Matches if argument is None or argument == other.

class AllOf(*c1*, *c2*, *...*)
Matches if argument satisfies all checkers c1, c2, ...

class Dict(*key_checker*, *value_checker*)
Matches if argument is a Mapping subtype, its keys all satisfy key_checker, and its values all satisfy value_checker.

class Union(*c1*, *c2*, *...*)
Matches if argument satisfies all of the checkers c1, c2, ...

class **Seq**(*checker*)
Matches if argument is a Sequence subtype whose elements all satisfy checker.

class **Set**(*checker*)
Matches if argument is a Set subtype whose elements all satisfy checker.

class **IsIterable**(*checker*)
Matches if argument is an Iterable subtype whose elements all satisfy checker.

class **Eq**(*other*)
matches if argument == other

class **Neq**(*other*)
matches if argument != other

class **Lt**(*other*)
matches if argument < other

class **Leq**(*other*)
matches if argument <= other

class **Gt**(*other*)
matches if argument > other

class **Geq**(*other*)
matches if argument >= other

10.3 codeviking.contracts.error Module

CHAPTER 11

Indices and tables

- *genindex*
- *modindex*
- *search*

C