PyContracts
@ Boston Python User Group
October 29, 2014

Andrea Censi

censi@mit.edu
http://censi.mit.edu

Other contributors to PyContracts: Brett Graham, William Furr, Karol Kuczmariski, Maarten Derickx, Calen Pennington, Adam Palay, Ryan Heimbuch, Bernhard Biskup.
Introduction

- Basic examples
- Defining contracts: decorators, docstrings, annotations
- Why using PyContracts?

Basic usage

- The contracts domain-specific language
- Useful constructs: logical operators, $\text{list}$, $\text{dict}$. 
- Disabling PyContracts in production

Advanced topics

- Extending PyContracts with $\text{new}\_\text{contract}$
- Variable binding - $\text{def f(x:'list[N]') -> 'list[N]' }$
- The $\text{ContractsMeta}$ metaclass
- The type operator - $\text{def f(x:'list(type(t))') -> 'type(t)' }$
The Basic Example

• Define a “contract” using the @contract decorator:

```python
from contracts import contract

@contract(x='int,>=0')
def f(x):
    pass
```

• Calling with an invalid argument raises an exception:

```python
>>> f(-2)
ContractNotRespected: Breach for argument 'x' to f(). Condition -2 >= 0 not respected
cHECKING: >=0 for value: Instance of int: -2
cHECKING: int,>=0 for value: Instance of int: -2
```
Contracts on Return Values

- Use the “returns” keyword for constraints on the return value.

```python
@contract(returns='int,>=0')
def f(x):
    return x
```

```python
>>> f(-2)
ContractNotRespected: Breach for return value of f().
Condition  -2  >=  0  not respected
checking:  >=0  for value: Instance of int:  -2
checking:  int,>=0  for value: Instance of int:  -2
```
3 Ways To Add Contracts

1. Using decorator arguments:

```python
@contract(x='int,>=0', returns='>=1')
def f(x):
    ...
```

2. Using docstrings:

```python
@contract
def f(x):
    """ Function description. 
    :type x: int,>0 
    :rtype: <=1 
    """
```

3. Using annotations (Python 3):

```python
@contract
def f(x:'int,>=0') -> '>=1':
    ...
```
Why using PyContracts?

• “Informal” documentation: Python is very concise, so you forget what were the assumptions you had.

```python
@contract(lines='list(str)',
          returns='dict(str: (int,>=1) )')
def word_count(lines):
    result = {}
    for line in lines:
        for word in line.split():
            result[word] = result.get(word, 0) + 1
    return result
```

• You can provide very informative errors, without writing any error checking code.

```python
>>> word_count([‘A’,’A’,1,’C’]
ContractNotRespected: Breach for argument ‘words’ to word_count().
Expected an ANSI string, got ‘int’.
checking: str         for value: Instance of int: 1
checking: list(str)   for value: Instance of list: ['A', 'A', 1, 'C']
```
Introduction

• Basic examples
• Defining contracts: decorators, docstrings, annotations
• Why using PyContracts?

Basic usage

• The contracts domain-specific language
• Useful constructs: logical operators, list, dict.
• Disabling PyContracts in production

Advanced topics

• Extending PyContracts with new_contract
• Variable binding - def f(x:'list[N]') -> 'list[N]'
• The ContractsMeta metaclass
• The type operator - def f(x:'list(type(t))') -> 'type(t)'
The Contracts language

- PyContracts uses a domain-specific language defined using PyParsing

- FAQ: Why not use Python directly?
  - Awkward for complicated expression
  - Need to import symbols.

### MyPy’s Syntax (a static type checker)

```python
from typing import Dict, List

def f(x: Dict[str, int]()) -> List[int]:
    ...
```

### PyContracts

```python
@contract
def f(x: 'dict(str:int)') -> 'list(int)':
    ...
```
Contracts language: AND operator

- Use , (comma) for AND

```python
@contract(prob='>=0,<=1')
def f(prob):
    pass
```

```python
>>> f(-1)
ContractNotRespected: Breach for argument 'prob' to f().
Condition -1 >= 0 not respected
checking: >=0 for value: Instance of int: -1
checking: >=0,<=1 for value: Instance of int: -1
```
Contracts language: OR operator

- Use | for OR

```python
@contract(x='<0|>1')
def f(x):
    ...
```

>>> f(2)
ContractNotRespected: Breach for argument 'x' to f().
Could not satisfy any of the 2 clauses in <0|>1.
    ---- Clause #1:  <0
    | Condition 0.5 < 0 not respected
    | checking: <0 for value: Instance of float: 0.5
    ---- Clause #2:  >1
    | Condition 0.5 > 1 not respected
    | checking: >1 for value: Instance of float: 0.5
    ------- (end clauses) -------
checking: <0|>1 for value: Instance of float: 0.5
Shortcuts

• Any Python type can be used as a contract:

```python
@contract(x=int)
def f(x):
    ...

@contract(x=MyClass)
def f(x):
    ...
```

• Strings are necessary for complicated expressions:

```python
@contract(x='int,>=0')
def f(x):
    ...

@contract(x='(int|float),>=0')
def f(x):
    ...
```
Contracts language: lists

• Lists have contracts for length and for the contained objects:

\[ \text{list}[\text{length contract}](\text{type contract}) \]

• A non-empty list \( \text{list}[>0] \)
• A list of integers \( \text{list}(\text{int}) \)
• A list of positive integers \( \text{list}(\text{int},>0) \)
• A non-empty list of positive integers \( \text{list}[>0](\text{int},>0) \)
Contracts language: dicts

- Dicts have contracts for size and for the keys and values

\[ \text{dict}[\text{length contract}](\text{key contract} : \text{value contract}) \]

- A dictionary
- A nonempty dictionary
- A dictionary whose keys are strings
- A dictionary whose keys are strings and the values are positive integers
Disabling PyContracts

- Two alternatives to disable all checks:
  1. Call `contracts.disable_all()`.
  2. Use the environment variable `DISABLE_CONTRACTS`.

- Performance impact:
  - If contracts are disabled before `@contract` is called, the function is returned undecorated ⇒ zero overhead
  - If contracts are disabled after `@contract` is called, the decorator does the equivalent of calling `f(*args, **kwargs)` ⇒ slight overhead

help?
**Introduction**

- Basic examples
- Defining contracts: decorators, docstrings, annotations
- Why using PyContracts?

**Basic usage**

- The contracts domain-specific language
- Useful constructs: logical operators, `list`, `dict`
- Disabling PyContracts in production

**Advanced topics**

- Extending PyContracts with `new_contract`
- Variable binding - `def f(x:'list[N]') -> 'list[N]'`
- The `ContractsMeta` metaclass
- The type operator - `def f(x:'list(type(t))') -> 'type(t)'`
Defining new contracts

• Use `new_contract` to define new contracts:

```python
from contracts import contract, new_contract

@new_contract
def even(x):
    if x % 2 != 0:
        msg = 'The number %s is not even.' % x
        raise ValueError(msg)

@contract(x='int,even')
def f(x):
    pass
```

```sh
>>> f(1)
contracts.interface.ContractNotRespected: Breach for argument 'x' to f(). The number 1 is not even.
checking: function even() for value: Instance of int: 1
checking: even for value: Instance of int: 1
checking: int,even for value: Instance of int: 1
```
Variable binding

- a b c d ... variables that bind to any object
- A B C D ... variables that bind to integers

```python
@contract(y='list[N](str),N>0',
          returns='list[N](>=0)')
def word_len(y):
    return map(len, y)
```

```python
>>> print word_len([])
ContractNotRespected: Breach for argument 'y' to word_len().
Condition Ø > Ø not respected
checking: N>Ø             for value: Instance of list: []
checking: list[N](str),N>Ø   for value: Instance of list: []
Variables bound in inner context:
- N: Instance of int: Ø
```
The ContractsMeta metaclass

- Define the contracts in the base class, without cluttering the derived class.

```python
from contracts import contract, ContractsMeta

class Base(object):
    __metaclass__ = ContractsMeta

    @abstractmethod
    @contract(probability='float,>=0,<=1')
    def sample(self, probability):
        pass

class Derived(Base):
    # The contract above is automatically enforced,
    # without this class having to know about PyContracts at all!
    def sample(self, probability):
        ....
```
The type operator

- Bind the variable `var` to the type of the object:
  
  \[ \text{type}(\text{var}) \]

- A list of elements of the same type:

```python
@contract(x='list(type(t))')
def fsum(x):
    return sum(x)
```

```python
>>> f([0, 1.0])
ContractNotRespected: Breach for return value of fmin().
Expected value for 't' was: Instance of type: <type 'int'>
    instead I received: Instance of type: <type 'float'>
checking: t         for value: Instance of type: <type 'float'>
checking: type(t)   for value: Instance of float: 1.0
Variables bound in inner context:
- t: Instance of type: <type 'int'>
```
The type operator

- Bind the variable `var` to the type of the object:
  
  \[ \text{type}(\text{var}) \]

- A list of elements of at most two different types:
  
  ```python
  @contract(x='list(type(t)|type(u))')
  def f(x):
      ...
  ```

```python
>>> f([1, 2, 'three', 'four', 2.1])
ContractNotRespected: Breach for argument 'x' to f().
Could not satisfy any of the 2 clauses in type(t)|type(u).
    ---- Clause #1:  type(t)
    | Expected value for 't' was: Instance of type: <type 'int'>
    | instead I received: Instance of type: <type 'float'>
    | checking: t for value: Instance of type: <type 'float'>
    | checking: type(t) for value: Instance of float: 2.1
    ---- Clause #2:  type(u)
    | Expected value for 'u' was: Instance of type: <type 'str'>
    | instead I received: Instance of type: <type 'float'>
    | checking: u for value: Instance of type: <type 'float'>
    | checking: type(u) for value: Instance of float: 2.1
    ------- (end clauses) -------
    checking: type(t)|type(u) for value: Instance of float: 2.1
    checking: list(type(t)|type(u)) for value: Instance of list: [1, 2, 'three', 'four', 2.1]
```

Variables bound in inner context:
- t: Instance of type: <type 'int'>
- u: Instance of type: <type 'str'>
Thanks!