Lecture 07: Exception Handling, Private and Managed Attributes

MPCS 51042-2 Fall 2019: Python Programming

Ron Rahaman

The University of Chicago, Dept of Computer Science

Nov 18, 2019

@classmethod因为持有cls参数，可以来调用类的属性，类的方法，实例化对象等

一般来说，要使用某个类的方法，需要先实例化一个对象再调用方法。

而使用@staticmethod或@classmethod，就可以不需要实例化，直接类名.方法名()来调用

Table of Contents

Table of Contents

What and Why Exception Handling

With exception handling, you can respond to runtime errors with custom-defined actions:

- Proceed with execution by correcting or ignoring error
- Halt execution and do necessary cleanup

In Python, exceptions are classes and part of an inheritance hierarchy

Exceptions from Built-in and User Objects

- Python built-ins raise exceptions such as `StopIteration`, `IndexError`, `AttributeError`, etc., based on particular events.
  - Normally, a built-in exception will halt execution and may not do necessary cleanup.
  - You can catch specific exceptions and specify exactly how/if to continue.
  - Reference: `https://docs.python.org/3/library/exceptions.html`
- In your objects, you can also raise built-in and user-defined exceptions whereever you want.

Table of Contents

Structure of a `try/except` Block

one try
one exception

```
try:
    # Main actions
except Exception1:
    # Executed when Exception1 is raised
except Exception2:
    # (Optionally-many) Executed when Exception2 is raised
else:
    # (Optional) Excecuted if no exception is raised
finally:
    # (Optional) Always run (whether any or no exception is raised)
# Execution returns here unless process exits
```

Structure of a `try/except` Block

1. The `try` block is executed
2. If exception is raised in `try` block and matches one named in an `except` block
   - 2.1 Execution halts in `try` at point where exception occurs.
   - 2.2 Execution continues in `except` block with matching exception
3. If no exceptions are raised in `try` block:
   - 3.1 Execution continues in `else` block
4. Execution continues in `finally` if exception was raised, if exception was not raised, or if program terminates itself.
5. If program hasn't terminated, exection continues outside of `try`/ `except` / `finally`

When in `finally` not executed?

Some system-level signals and errors can terminate the Python program without running the `finally` block.

- `SIGKILL`
- Some situations with zombie processes and threads
- Power loss
- etc.

Avoid bare `except`!

- Python allows a bare `except` block to catch any exception that is not named
- This is bad form!
- Some system errors should always be handled by Python, not you!
- Best practice (more on this later): `except Exception:`

```
try:
    # Main actions
except Exception1:
    # Executed when Exception1 is raised
except Exception2:
    # (Optionally-many) Executed when Exception2 is raised
except:
    # Executed for any unspecified exceptions
```

Table of Contents

Base Classes for Built-in Exceptions

Exceptions have a shallow class hierarchy (`https://docs.python.org/3/library/exceptions.html#exception-hierarchy`)

The two main superclasses are:

- `BaseException`:
  - Base class of all exceptions.
  - Provides printing, constructor, state retention.
- `Exception`:
  - Immediate subclass of Exception.
  - Superclass of all other built-in exceptions other than system exits.
  - Virtually all user classes should be derived from `Exception` instead of `BaseException`.

Exception can be accessed as object, But it doesn't catch BaseException or the system-exiting exceptions SystemExit, KeyboardInterrupt and GeneratorExit:

BaseException vs. Exception

The distinction between `Exception` and `BaseException` allows us to catch all user-space errors while still allowing system errors to halt execution.

- System errors (derived from `BaseException`) should normally be allowed to kill your program. Examples: `SystemExit`, `KeyboardInterrupt`.
- Using `except Exception` is almost always preferable to bare `except`

```
try:
    # Do something
except Exception:
    # Handle user-space errors
```

Some Useful Built-In Exceptions

- `ArithmeticError`: Base class for various arithmetic errors: (`ZeroDivisionError`, `OverflowError`, etc.)
- `AttributeError`: An attribute reference or assignment fails.
- `LookupError`: Base class for invalid key or index
  - `IndexError`: When a sequence subscript is out-of-range
  - `KeyError`: When a mapping (dict) key is not found
- `TypeError`: When an operator/function is given an operand/argument of the **incorrect type**.
- `ValueError`: When an operator/function is given an operand/argument of the **correct type but incorrect value**.

Table of Contents

Raising Exceptions

The `raise` statement allows you to raise exceptions explicitly.
You can raise both:

- An `Exception` instance: Optional first constructor arg is an error message
- An `Exception` class: An instance with no constructor args is created.

```
# Explcitly create instance
raise ValueError("Oops!")
# Implicily create instance.  Same as raise ValueError()
raise ValueError
```

Re-raising Exceptions

After catching an exception in a `try` block, the exception can be re-raised in two ways:

- When using `catch SomeExeption as e:`, the instance of the exception is assigned to `e`
  - The instance has a few attributes: `https://docs.python.org/3/library/exceptions.html#BaseException`
  - The instance can be re-raised: `raise e`
- Using `raise` without specifying an exception will re-raise the last exception that was caught.

Re-raising Exception

What happens here?

```
L = []
try:
    L[0] = 1/0
except Exception as e:
    print("OOPS!")
    raise e
```

oops,

division by zero error

Table of Contents

Example: Opening a File

This program will crash if the first file is missing. You can't continue to process the existing file.

```
linelist = []

for fname in ['missing_file.txt', 'existing_file.txt']:
    with open(fname) as f:
        linelist.extend(f.readlines())

print(linelist)
```

UNIX 3 reserved :stdinput,stdoutput,stderror.
Standard error, abbreviated stderr, is the destination of error messages from command line

Example: Opening a File, cont

- In general, we surround **as small a region of code as possible** with the try/except.
- Here, we only want to ignore `FileNotFoundError` and allow other errors to stop the program.
- Best practice to print warnings to stderr, not stdout

if the file is missing, we still want to import other files

```
import sys
linelist = []

for fname in ['missing_file.txt', 'lecture07.ipynb']:
    try:
        with open(fname) as f:
            linelist.extend(f.readlines())
    except FileNotFoundError:
        print("'{}' doesn't exist!".format(fname), file=sys.stderr)
```

Exception Instances

- An exception creates an instance of an exception object.
- `except Exception as var_name` allows us to refer to that instance
- The instance can be printed

```
import sys
linelist = []

for fname in ['missing_file.txt', 'Untitled.ipynb']:
    try:
        with open(fname) as f:
            linelist.extend(f.readlines())
    except FileNotFoundError as e:
        print(e, file=sys.stderr)
```

No such file: missing.....

Warnings

- The stdlib warnings module provides high-level support for non-fatal warnings
- When running the code, the user can choose to filter warnings and make specific warnings fatal
- https://docs.python.org/3/library/warnings.html

```
import warnings
linelist = []

for fname in ['missing_file.txt', 'Untitled.ipynb']:
    try:
        with open(fname) as f:
            linelist.extend(f.readlines())
    except FileNotFoundError as e:
        warnings.warn(e)
```

pass the instance of exception

Multiple Exceptions in One Block

- Both FileNotFoundError and PermissionError will be handled by the same **except** block
- e will refer to particular instance that was raised

```
import warnings
linelist = []

for fname in ['missing_file.txt', 'Untitled.ipynb']:
    try:
        with open(fname) as f:
            linelist.extend(f.readlines())
    except (FileNotFoundError, PermissionError) as e:
        warnings.warn(e)
```

could make a tuple of many errors

Example: HTTP requests

- The `requests` module is a very well-accepted third-party library for making HTTP requests
  https://requests.readthedocs.io/en/master/
- Example: this request could possibly hang, since `requests.get()` doesn't have a default timeout.

```
import requests
r = requests.get('https://api.github.com/events',
                 auth=('RonRahaman', 'foobar'))
response = r.json()
print("response = {}".format(response))
```

Example: HTTP requests, cont.

- Best practices: Use `timeout` argument in `requests.get()`,
- This raises `requests.exceptions.ConnectTimeout`

```
import warnings
try:
    r = requests.get('https://api.github.com/events',
                     auth=('RonRahaman', 'foobar'),
                     timeout=0.0001)
except requests.exceptions.ConnectTimeout as e:
    warnings.warn(e)
    response = {}
else:
    response = r.json()
```

GET is used to request data from a specified resource.

Example: HTTP requests, cont.

- `requests` has an object hierarchy for its exceptions
- `ConnectTimeout` is a subclass of `Timeout`, along with other kinds of timeout exceptions
- If desired, we could catch more kinds of timeout exceptions

```
try:
    r = requests.get('https://api.github.com/events',
                     auth=('RonRahaman', 'foobar'),
                     timeout=0.0001)
except requests.exceptions.Timeout as e:
    warnings.warn(e)
    response = {}
else:
    response = r.json()
```

Example: Using exceptions for polymorphism

- This Fraction takes any args that can be converted to int
- Is the try/except strictly necessary?
- Do you think this is too broadly-defined?

```
class Fraction:
    def __init__(self, numerator, denominator):
        try:
            self.numerator = int(numerator)
            self.denominator = int(denominator)
        except ValueError:
            raise ValueError("{} constructor expects an integer" \
                             "numerator and denominator".format(
                             self.__class__.__name__))

    def __repr__(self):
        return f'Fraction({self.numerator}, {self.denominator})'
```

Example: Using exceptions for polymorphism, cont.

```
>>> Fraction(1, 2)
Fraction(1, 2)
>>> Fraction('1', '2')
Fraction(1, 2)
>>> Fraction(1.1, 2.0)  # Hmmm... :/
Fraction(1, 2)
>>> Fraction('foo', 'bar')
Traceback (most recent call last):
  File "<stdin>", line 4, in __init__
ValueError: invalid literal for int() with base 10: 'foo'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in __init__
ValueError: Fraction constructor expects an integer numerator and denominat
```

Example: Using exceptions for polymorphism, cont

- This is more specific and returns a user-friendly exception.

```
from numbers import Integral

class Fraction:
    def __init__(self, numerator, denominator):
        if isinstance(numerator, Integral) and \
                isinstance(denominator, Integral):
            self.numerator = numerator
            self.denominator = denominator
        else:
            raise TypeError("{} constructor expects an Integral " \
                            "numerator and denominator".format(
                            self.__class__.__name__))

    def __repr__(self):
        return f'Fraction({self.numerator}, {self.denominator})'
```

type check first

little limited

Example: Using exceptions for polymorphism, cont.

```
>>> Fraction(1, 2)
Fraction(1, 2)
>>> Fraction('1', '2')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in __init__
TypeError: Fraction constructor expects an Integral numerator and denominat
>>> Fraction(1.0, 2.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in __init__
TypeError: Fraction constructor expects an Integral numerator and denominat
>>> f1 = Fraction(1, 3)
>>> Fraction(2*f1.numerator, f1.denominator)
Fraction(2, 3)
```

Example: Using exceptions for polymorphism, cont.

- This works for any object with a numerator and denominator
- Still pretty specific

```python
class Fraction:
    # Same constructor and __repr__ as before

    def __mul__(self, other):
        try:
            return Fraction(self.numerator * other.numerator,
                            self.denominator * other.denominator)
        except AttributeError:
            return NotImplemented

    def __rmul__(self, other):
        return self * other
```

Example: Using exceptions for polymorphism, cont.

```
>>> Fraction(1, 2) * Fraction(1, 3)
Fraction(1, 6)
>>> 2 * Fraction(1,3)
Fraction(2, 3)
>>> 2.0 * Fraction(1,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 17, in __rmul__
TypeError: unsupported operand type(s) for *: 'Fraction' and 'float'
```

Notimplemented only has one object

verloading is about same function have different signatures. Overriding is about same function, same signature but different classes connected through inheritance.

overloading :same function different parameters

Table of Contents

Table of Contents

Private Attributes: Safety vs. Security

- There are no "private" or "protected" declarations in Python.
- Even in C++ and Java, private attributes only provide:
  - Safety against programming errors from other developers.
  - NOT security against attack by adversaries.
- Likewise, privates prevent:
  - Accidental misuse
  - NOT intentional misuse

C++: Intentional Misuse of Privates

- The compiler can detect access to private attributes by name
- In this case, the compiler cannot detect access by pointer.

```cpp
class PubPriv {
public:
  int pub = 1;
private:
  int priv = 123456;
};

int main() {
  PubPriv foo;

  // foo.pub can be accessed by name
  cout << "Public: " << foo.pub << endl;

  // foo.priv can be accessed by offset relative to foo.pub
  char* ptr_to_priv = (char*) &foo.pub + sizeof(int);
  cout << "Private: " << *((int*) ptr_to_priv) << endl;
}
```

Java: Intentional Misuse of Privates

From *Fluent Python*, Ch 9

1. First, define a class with private attribute and a public mutator

```
public class Confidential {
    private String secret = "";
    public Confidential(String text) { secret = text.toUpperCase(); }
}
```

2. Then use well-known introspection interface to access privates. Can be done in pure Java, but done here in Jython for brevity.

```
import Confidential

message = Confidential('top secret text')
secret_field = Confidential.getDeclaredField('secret')
secret_field.setAccessible(True)  # break the lock!
print 'message.secret =', secret_field.get(message)
```

Refer to Java resources for more details.

Table of Contents

Example: A Mutex Object

- To demonstrate different techniques for attribute access, we will implement a very basic mutex.
- Our `Mutex` class will have the following methods:
  - `lock(caller_id)`: If the mutex is currently unlocked, then the caller acquires/locks the mutex. Otherwise, nothing happens.
  - `unlock(caller_id)`: If the caller currently owns the mutex, then the mutex is unlocked. Otherwise, nothing happens.
  - `is_owned(caller_id)`: Returns `True` if the caller owns the mutex.
  - `is_locked()`: Returns `True` if the mutex is locked by anyone.

mutex is to manage shared resource

Example: A Mutex Object

```
class Mutex:
    def __init__(self):
        self._owner = None

    def is_locked(self):
        return (self._owner != None)

    def is_owner(self, caller):
        return (self._owner == caller)

    def lock(self, caller):
        if not self.is_locked():
            self._owner = caller

    def unlock(self, caller):
        if self.is_owner(caller):
            self._owner = None

    def __repr__(self):
        return "Mutex(owner={})".format(self._owner)
```

set the owner

acquire

release

single underscore:
The underscore prefix is meant as a hint to another programmer that a variable or method starting with a single underscore is intended for internal use.

Convention 1: Single Leading Underscore

Attributes named with a single leading underscore (e.g., `_owner`) can be used to indicate private variables.

- Very widespread Python programming convention.
- Tells other programmers that attributes should not be accessed outside of class.
- No significance to Python itself:

```
>>> m = Mutex()
>>> m.lock(1)          # Mutex(owner=1)
>>> m.lock(2)          # Correct interface. Mutex(owner=1)
>>> m._owner = 2       # Broke interface!   Mutex(owner=2)
```

Example: The MutexVar Class

- Now we implement a derived class for a mutex with an associated, arbitrarily-valued variable.
- Our MutexVar class will have the following methods:
  - Inherited methods from `Mutex`
  - `get(caller_id)`: If the `MutexVar` is currently unlocked **or** the caller currently owns it, then return the `MutexVar`'s current value. Otherwise, do nothing.
  - `set(caller_id, value)`: If the caller currently owns the `MutexVar`, then set the new value. Otherwise, do nothing.
- When implementing `MutexVar`, we will not break the interface of `Mutex`.

Example: The MutexVar Class

```
class MutexVar(Mutex):

    def __init__(self):
        self._val = None
        Mutex.__init__(self)

    def get(self, caller):
        if not self.is_locked() or self.is_owner(caller):
            return self._val
        else:
            return None

    def set(self, caller, val):
        if self.is_owner(caller):
            self._val = val

    def __repr__(self):
        return "MutexVar(val={}, {})".format(self._val,
                Mutex.__repr__(self))
```

Example: The MutexVar Class

We use the interfaces provided by `Mutex` and `MutexVar`:

```
>>> v = MutexVar()          # MutexVar(val=None, Mutex(owner=None))
>>> v.lock(1)               # MutexVar(val=None, Mutex(owner=1))
>>> v.set(1, 'apple')       # MutexVar(val=apple, Mutex(owner=1))

>>> v.get(1)                # Caller 1 can get value
'apple'

>>> v.get(2)                # Caller 2 cannot get value until 1 unlocks it
>>> v.unlock(1)
>>> v.get(2)
'apple'
```

Table of Contents

Name Clashes with Private Variables

- Attributes of a subclass will override attributes with the same name in the base class.
- For private variables, this is often an error
- For example, suppose that:
  - `Mutex` required a private attribute called `_val`
  - Unaware of this, the implementer of `MutexVar` used an attribute called `_val` for a different purpose.

Name Clashes in Another Implementation

- This implementation of `Mutex` relies on a new private attribute, `_var`

```
class Mutex:
    def __init__(self):
        self._owner = None
        self._val = 'unlocked'

    def is_locked(self):
        if self._val == 'locked':
            return True
        elif self._val == 'unlocked':
            return False
        else:
            raise ValueError('Invalid value for mutex: "{}"'.format(self._v

    # ... see source code for more ...
```

- The implementation of `MutexVar` is the same as before.

Name Clashes in Another Implementation

- `v._val` is touched by methods from both `Mutex` and `MutexVar`:
- Below, an error occurs when `MutexVar.get()` calls `Mutex.is_locked()` and checks if `_val` is locked, unlocked, or invalid.

```
>>> v = MutexVar()
MutexVar(val=unlocked, Mutex(owner=None, val=unlocked))
>>> v.lock(1)
MutexVar(val=locked, Mutex(owner=1, val=locked))
>>> v.set(1, 'apple')
MutexVar(val=apple, Mutex(owner=1, val=apple))
>>> v.get(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in get
  File "<stdin>", line 12, in is_locked
ValueError: Invalid value for mutex: "apple"
```

Table of Contents

Leading double underscore names are private (meaning not available to derived classes)

Name-Mangling with Leading Double-Underscores

- For variables starting with a leading double-underscore, the Python interpreter performs an extra name-mangling step.
  - Outside the class definition, these attributes are only accessible via a mangled (modified) name
  - The name mangling scheme is always: `_classname__attributename`
- Example: `__foo` is used normally inside the class def

```
class MyClass:
    def __init__(self, foo):
        self.__foo = foo
    def get_foo(self):
        return self.__foo
```

- Outside, it can only be accessed via its name-mangled version

```
>>> x = MyClass('banana')
>>> x.get_foo()
'banana'
>>> x._MyClass__foo
'banana'
```

Name-Mangling Prevents Name Clashes

- If we use double-underscored names in `Mutex` and `MutexVar`, then we can prevent the name clashes.
- Below, we have created two separate attributes for val:
  - In the respective class defs, they are available as `__val`
  - Outside off the defs, they are available as `_Mutex__val` and `_MutexVar__val`.

```
class Mutex:
    def __init__(self):
        self.__owner = None
        self.__val = 'unlocked'
    # ... etc ...

class MutexVar(Mutex):
    def __init__(self):
        self.__val = None
        Mutex.__init__(self)
    # ... etc ...
```

It will add the class name before it.
But the single underscore might be overwriten because of inheritance

Name-Mangling Prevents Name Clashes

The name-mangled attributes maintain separate states.

```
>>> v = MutexVar()
MutexVar(val=None, Mutex(owner=None, val=unlocked))
>>> v.lock(1)
MutexVar(val=None, Mutex(owner=1, val=locked))
>>> v.set(1, 'apple')
MutexVar(val=apple, Mutex(owner=1, val=locked))
>>> v.get(1)
'apple'
```

Table of Contents

Table of Contents

Welford's Algorithm

- Welford's algorithm is a numerically-stable, one-pass algorithm for calculating the mean and sums of squared differences (and hence variance and sample variance, too)
- `https://jonisalonen.com/2013/deriving-welfords-method-for-computing-variance/`

Updating Mean and SS

```
class Welfords:
    def __init__(self, *args):
        self._mean = 0
        self._ss = 0
        self._count = 0
        self.update(*args)

    def update(self, *args):
        for x in args:
            self._count += 1
            old_mean = self._mean
            self._mean += (x - self._mean) / self._count
            self._ss += (x - self._mean) * (x - old_mean)
```

Updating Mean and SS

```python
class Welfords:
    # Continued from above

    def get_mean(self):
        return self._mean

    def get_variance(self):
        return self._ss / self._count

    def get_sample_variance(self):
        return self._ss / (self._count - 1)
```

The `@property` decorator

- The `@property` decorator creates a managed attribute (or property) from a method.
- After decorating these methods, their results are accessible as read-only attributes

```python
class Welfords:
    # Re-using __init__() and update()

    @property
    def mean(self):
        return self._mean

    @property
    def variance(self):
        return self._ss / self._count

    @property
    def sample_variance(self):
        return self._ss / (self._count - 1)
```

The @property decorator

- The @property decorator creates a managed attribute (or property) from a method.
- After decorating these methods, their results are accessible as read-only attributes

```
class Welfords:
    # Re-using __init__() and update()

    @property
    def mean(self):
        return self._mean

    @property
    def variance(self):
        return self._ss / self._count

    @property
    def sample_variance(self):
        return self._ss / (self._count - 1)
```

The `@property` decorator, cont.

```
>>> w = Welfords()
>>> w.update(4)
>>> w.update(6)
>>> w.mean
5.0
>>> w.variance
1.0
>>> w.update(3)
>>> w.mean
4.333333333333333
>>> w.variance
1.5555555555555554
>>> w.mean = 1000
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Table of Contents

Read-write Properties

- The property protocol also allows mananged read/write properties:

```
class Sphere:

    def __init__(self, radius):
        self.__radius = radius

    @property
    def radius(self):
        return self.__radius

    @radius.setter
    def radius(self, rad):
        if rad > 0:
            self.__radius = rad
        else:
            raise ValueError("Radius must be non-negative.")
```

The method which has to function as the setter is decorated with "@x.setter".

Read-write Properties

- Translating the decorator syntax:

```
class Sphere:
    # same __init__ as before

    def get_radius(self):
        return self.__radius
    # "radius" is re-assigned to an instance of the "property" class
    radius = property(get_radius)

    def set_radius(self, rad):
        if rad > 0:
            self.__radius = rad
        else:
            raise ValueError("Radius must be non-negative.")
    # A method of the "property" class.
    # Creates a copy of "radius" with a new setter method,
    # then re-assigns it to "radius".
    radius = radius.setter(set_radius)
```

More Read-Write Properties for Spheres

we wrote "two" methods with the same name and a different number of parameters "def x(self)" and "def x(self,x)". We have learned in a previous chapter of our course that this is not possible. It works here due to the decorating:

- Can get and set volume without explicit attribute

```
import numpy as np
class Sphere:
    # same __init__ and radius() as before

    @property
    def volume(self):
        return 4/3 * np.pi * self.__radius**3

    @volume.setter
    def volume(self, vol):
        if vol > 0:
            self.__radius = np.cbrt(vol * 3 / (4 * np.pi))
        else:
            raise ValueError("Volume must be non-negative.")
```

More Read-Write Properties for Spheres

- Can get and set surface area without explicit attribute

```
import numpy as np
class Sphere:
    # same __init__, radius(), and volume() as before

    @property
    def area(self):
        return 4 * np.pi * self.__radius**2

    @area.setter
    def area(self, area):
        if area > 0:
            self.__radius = np.sqrt(area / 4 / np.pi)
        else:
            raise ValueError("Area must be non-negative")
```

like a error checking

Alternate Constructors for Spheres

- Can easily write constructors for volume and surface area

```
import numpy as np
class Sphere:
    # same __init__, radius(), volume(), and area()

    @classmethod
    def from_volume(cls, volume):
        s = cls(0)
        s.volume = volume
        return s

    @classmethod
    def from_area(cls, area):
        s = cls(0)
        s.area = area
        return s
```

Alternatively, we could have used a different syntax without decorators to define the property

https://www.python-course.eu/python3_properties.php

Validated Attributes (Ramalho Ch 19)

```
class LineItem:

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price

    @property
    def weight(self):
        return self.__weight

    @weight.setter
    def weight(self, value):
        if value > 0:
            self.__weight = value
        else:
            raise ValueError('value must be > 0')
```