9.1 在函数上添加包装器¶

问题¶

你想在函数上添加一个包装器，增加额外的操作处理(比如日志、计时等)。

解决方案¶

如果你想使用额外的代码包装一个函数，可以定义一个装饰器函数，例如：

```
import time
from functools import wraps

def timethis(func):
    '''
    Decorator that reports the execution time.
    '''
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end-start)
        return result
    return wrapper
```

下面是使用装饰器的例子：

```
>>> @timethis
... def countdown(n):
...     '''
...     Counts down
...     '''
...     while n > 0:
...         n -= 1
...
>>> countdown(100000)
countdown 0.008917808532714844
>>> countdown(10000000)
countdown 0.87188299392912
>>>
```

讨论¶

一个装饰器就是一个函数，它接受一个函数作为参数并返回一个新的函数。 当你像下面这样写：

```
@timethis
def countdown(n):
    pass
```

跟像下面这样写其实效果是一样的：

```
def countdown(n):
    pass
countdown = timethis(countdown)
```

顺便说一下，内置的装饰器比如 `@staticmethod`, `@classmethod`,`@property` 原理也是一样的。 例如，下面这两个代码片段是等价的：

```
class A:
    @classmethod
    def method(cls):
        pass
```

```
class B:
    # Equivalent definition of a class method
    def method(cls):
        pass
    method = classmethod(method)
```

在上面的 `wrapper()` 函数中， 装饰器内部定义了一个使用 `*args` 和 `**kwargs` 来接受任意参数的函数。 在这个函数里面调用了原始函数并将其结果返回，不过你还可以添加其他额外的代码(比如计时)。 然后这个新的函数包装器被作为结果返回来代替原始函数。

需要强调的是装饰器并不会修改原始函数的参数签名以及返回值。 使用 `*args` 和 `**kwargs` 目的就是确保任何参数都能适用。 而返回结果值基本都是调用原始函数 `func(*args, **kwargs)` 的返回结果，其中func就是原始函数。

刚开始学习装饰器的时候，会使用一些简单的例子来说明，比如上面演示的这个。 不过实际场景使用时，还是有一些细节问题要注意的。 比如上面使用 `@wraps(func)` 注解是很重要的， 它能保留原始函数的元数据(下一小节会讲到)，新手经常会忽略这个细节。 接下来的几个小节我们会更加深入的讲解装饰器函数的细节问题，如果你想构造你自己的装饰器函数，需要认真看一下。

9.10 为类和静态方法提供装饰器¶

问题¶

你想给类或静态方法提供装饰器。

解决方案¶

给类或静态方法提供装饰器是很简单的，不过要确保装饰器在 `@classmethod` 或 `@staticmethod` 之前。例如：

```
import time
from functools import wraps

# A simple decorator
def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        r = func(*args, **kwargs)
        end = time.time()
        print(end-start)
        return r
    return wrapper

# Class illustrating application of the decorator to different kinds of methods
class Spam:
    @timethis
    def instance_method(self, n):
        print(self, n)
        while n > 0:
            n -= 1

    @classmethod
    @timethis
    def class_method(cls, n):
        print(cls, n)
        while n > 0:
            n -= 1

    @staticmethod
    @timethis
    def static_method(n):
        print(n)
        while n > 0:
            n -= 1
```

装饰后的类和静态方法可正常工作，只不过增加了额外的计时功能：

```
>>> s = Spam()
>>> s.instance_method(1000000)
<__main__.Spam object at 0x1006a6050> 1000000
0.11817407608032227
>>> Spam.class_method(1000000)
<class '__main__.Spam'> 1000000
0.11334395408630371
>>> Spam.static_method(1000000)
1000000
0.11740279197692871
>>>
```

讨论¶

如果你把装饰器的顺序写错了就会出错。例如，假设你像下面这样写：

```
class Spam:
```

```
    @timethis
    @staticmethod
    def static_method(n):
        print(n)
        while n > 0:
            n -= 1
```

那么你调用这个静态方法时就会报错：

```
>>> Spam.static_method(1000000)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "timethis.py", line 6, in wrapper
start = time.time()
TypeError: 'staticmethod' object is not callable
>>>
```

问题在于 `@classmethod` 和 `@staticmethod` 实际上并不会创建可直接调用的对象， 而是创建特殊的描述器对象(参考8.9小节)。因此当你试着在其他装饰器中将它们当做函数来使用时就会出错。 确保这种装饰器出现在装饰器链中的第一个位置可以修复这个问题。

当我们在抽象基类中定义类方法和静态方法(参考8.12小节)时，这里讲到的知识就很有用了。 例如，如果你想定义一个抽象类方法，可以使用类似下面的代码：

```
from abc import ABCMeta, abstractmethod
class A(metaclass=ABCMeta):
    @classmethod
    @abstractmethod
    def method(cls):
        pass
```

在这段代码中，`@classmethod` 跟 `@abstractmethod` 两者的顺序是有讲究的，如果你调换它们的顺序就会出错。

9.11 装饰器为被包装函数增加参数¶

问题¶

你想在装饰器中给被包装函数增加额外的参数，但是不能影响这个函数现有的调用规则。

解决方案¶

可以使用关键字参数来给被包装函数增加额外参数。考虑下面的装饰器：

```
from functools import wraps

def optional_debug(func):
    @wraps(func)
    def wrapper(*args, debug=False, **kwargs):
        if debug:
            print('Calling', func.__name__)
        return func(*args, **kwargs)

    return wrapper
```

```
>>> @optional_debug
... def spam(a,b,c):
...     print(a,b,c)
...
>>> spam(1,2,3)
1 2 3
>>> spam(1,2,3, debug=True)
Calling spam
1 2 3
>>>
```

讨论¶

通过装饰器来给被包装函数增加参数的做法并不常见。 尽管如此，有时候它可以避免一些重复代码。例如，如果你有下面这样的代码：

```
def a(x, debug=False):
    if debug:
        print('Calling a')

def b(x, y, z, debug=False):
    if debug:
        print('Calling b')

def c(x, y, debug=False):
    if debug:
        print('Calling c')
```

那么你可以将其重构成这样：

```
from functools import wraps
import inspect

def optional_debug(func):
    if 'debug' in inspect.getargspec(func).args:
        raise TypeError('debug argument already defined')

    @wraps(func)
    def wrapper(*args, debug=False, **kwargs):
        if debug:
            print('Calling', func.__name__)
        return func(*args, **kwargs)
    return wrapper
```

```
@optional_debug
def a(x):
    pass

@optional_debug
def b(x, y, z):
    pass

@optional_debug
def c(x, y):
    pass
```

这种实现方案之所以行得通，在于强制关键字参数很容易被添加到接受 `*args` 和 `**kwargs` 参数的函数中。 通过使用强制关键字参数，它被作为一个特殊情况被挑选出来， 并且接下来仅仅使用剩余的位置和关键字参数去调用这个函数时，这个特殊参数会被排除在外。 也就是说，它并不会被纳入到 `**kwargs` 中去。

还有一个难点就是如何去处理被添加的参数与被包装函数参数直接的名字冲突。 例如，如果装饰器 `@optional_debug` 作用在一个已经拥有一个 `debug` 参数的函数上时会有问题。 这里我们增加了一步名字检查。

上面的方案还可以更完美一点，因为精明的程序员应该发现了被包装函数的函数签名其实是错误的。例如：

```
>>> @optional_debug
... def add(x,y):
...     return x+y
...
>>> import inspect
>>> print(inspect.signature(add))
(x, y)
>>>
```

通过如下的修改，可以解决这个问题：

```
from functools import wraps
import inspect

def optional_debug(func):
    if 'debug' in inspect.getargspec(func).args:
        raise TypeError('debug argument already defined')

    @wraps(func)
    def wrapper(*args, debug=False, **kwargs):
        if debug:
            print('Calling', func.__name__)
        return func(*args, **kwargs)

    sig = inspect.signature(func)
    parms = list(sig.parameters.values())
    parms.append(inspect.Parameter('debug',
                inspect.Parameter.KEYWORD_ONLY,
                default=False))
    wrapper.__signature__ = sig.replace(parameters=parms)
    return wrapper
```

通过这样的修改，包装后的函数签名就能正确的显示 `debug` 参数的存在了。例如：

```
>>> @optional_debug
... def add(x,y):
...     return x+y
...
>>> print(inspect.signature(add))
(x, y, *, debug=False)
>>> add(2,3)
5
>>>
```

参考9.16小节获取更多关于函数签名的信息。

9.12 使用装饰器扩充类的功能¶

问题¶

你想通过反省或者重写类定义的某部分来修改它的行为，但是你又不希望使用继承或元类的方式。

解决方案¶

这种情况可能是类装饰器最好的使用场景了。例如，下面是一个重写了特殊方法 `__getattribute__` 的类装饰器， 可以打印日志：

```
def log_getattribute(cls):
    # Get the original implementation
    orig_getattribute = cls.__getattribute__

    # Make a new definition
    def new_getattribute(self, name):
        print('getting:', name)
        return orig_getattribute(self, name)

    # Attach to the class and return
    cls.__getattribute__ = new_getattribute
    return cls

# Example use
@log_getattribute
class A:
    def __init__(self,x):
        self.x = x
    def spam(self):
        pass
```

下面是使用效果：

```
>>> a = A(42)
>>> a.x
getting: x
42
>>> a.spam()
getting: spam
>>>
```

讨论¶

类装饰器通常可以作为其他高级技术比如混入或元类的一种非常简洁的替代方案。 比如，上面示例中的另外一种实现使用到继承：

```
class LoggedGetattribute:
    def __getattribute__(self, name):
        print('getting:', name)
        return super().__getattribute__(name)

# Example:
class A(LoggedGetattribute):
    def __init__(self,x):
        self.x = x
    def spam(self):
        pass
```

这种方案也行得通，但是为了去理解它，你就必须知道方法调用顺序、`super()` 以及其它8.7小节介绍的继承知识。 某种程度上来讲，类装饰器方案就显得更加直观，并且它不会引入新的继承体系。它的运行速度也更快一些， 因为他并不依赖 `super()` 函数。

如果你想在一个类上面使用多个类装饰器，那么就需要注意下顺序问题。 例如，一个装饰器A会将其装饰的方法完整

替换成另一种实现， 而另一个装饰器B只是简单的在其装饰的方法中添加点额外逻辑。 那么这时候装饰器A就需要放在装饰器B的前面。

你还可以回顾一下8.13小节另外一个关于类装饰器的有用的例子。

9.13 使用元类控制实例的创建¶

问题¶

你想通过改变实例创建方式来实现单例、缓存或其他类似的特性。

解决方案¶

Python程序员都知道，如果你定义了一个类，就能像函数一样的调用它来创建实例，例如：

```
class Spam:
    def __init__(self, name):
        self.name = name

a = Spam('Guido')
b = Spam('Diana')
```

如果你想自定义这个步骤，你可以定义一个元类并自己实现 `__call__()` 方法。

为了演示，假设你不想任何人创建这个类的实例：

```
class NoInstances(type):
    def __call__(self, *args, **kwargs):
        raise TypeError("Can't instantiate directly")

# Example
class Spam(metaclass=NoInstances):
    @staticmethod
    def grok(x):
        print('Spam.grok')
```

这样的话，用户只能调用这个类的静态方法，而不能使用通常的方法来创建它的实例。例如：

```
>>> Spam.grok(42)
Spam.grok
>>> s = Spam()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example1.py", line 7, in __call__
        raise TypeError("Can't instantiate directly")
TypeError: Can't instantiate directly
>>>
```

现在，假如你想实现单例模式（只能创建唯一实例的类），实现起来也很简单：

```
class Singleton(type):
    def __init__(self, *args, **kwargs):
        self.__instance = None
        super().__init__(*args, **kwargs)

    def __call__(self, *args, **kwargs):
        if self.__instance is None:
            self.__instance = super().__call__(*args, **kwargs)
            return self.__instance
        else:
            return self.__instance

# Example
class Spam(metaclass=Singleton):
    def __init__(self):
        print('Creating Spam')
```

那么Spam类就只能创建唯一的实例了，演示如下：

```
>>> a = Spam()
```

```
Creating Spam
>>> b = Spam()
>>> a is b
True
>>> c = Spam()
>>> a is c
True
>>>
```

最后，假设你想创建8.25小节中那样的缓存实例。下面我们可以通过元类来实现：

```
import weakref

class Cached(type):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__cache = weakref.WeakValueDictionary()

    def __call__(self, *args):
        if args in self.__cache:
            return self.__cache[args]
        else:
            obj = super().__call__(*args)
            self.__cache[args] = obj
            return obj

# Example
class Spam(metaclass=Cached):
    def __init__(self, name):
        print('Creating Spam({!r})'.format(name))
        self.name = name
```

然后我也来测试一下：

```
>>> a = Spam('Guido')
Creating Spam('Guido')
>>> b = Spam('Diana')
Creating Spam('Diana')
>>> c = Spam('Guido') # Cached
>>> a is b
False
>>> a is c # Cached value returned
True
>>>
```

讨论¶

利用元类实现多种实例创建模式通常要比不使用元类的方式优雅得多。

假设你不使用元类，你可能需要将类隐藏在某些工厂函数后面。 比如为了实现一个单例，你你可能会像下面这样写：

```
class _Spam:
    def __init__(self):
        print('Creating Spam')

_spam_instance = None

def Spam():
    global _spam_instance

    if _spam_instance is not None:
        return _spam_instance
    else:
        _spam_instance = _Spam()
        return _spam_instance
```

尽管使用元类可能会涉及到比较高级点的技术，但是它的代码看起来会更加简洁舒服，而且也更加直观。

更多关于创建缓存实例、弱引用等内容，请参考8.25小节。

9.14 捕获类的属性定义顺序¶

问题¶

你想自动记录一个类中属性和方法定义的顺序， 然后可以利用它来做很多操作（比如序列化、映射到数据库等等）。

解决方案¶

利用元类可以很容易的捕获类的定义信息。下面是一个例子，使用了一个OrderedDict来记录描述器的定义顺序：

```
from collections import OrderedDict

# A set of descriptors for various types
class Typed:
    _expected_type = type(None)
    def __init__(self, name=None):
        self._name = name

    def __set__(self, instance, value):
        if not isinstance(value, self._expected_type):
            raise TypeError('Expected ' + str(self._expected_type))
        instance.__dict__[self._name] = value

class Integer(Typed):
    _expected_type = int

class Float(Typed):
    _expected_type = float

class String(Typed):
    _expected_type = str

# Metaclass that uses an OrderedDict for class body
class OrderedMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        d = dict(clsdict)
        order = []
        for name, value in clsdict.items():
            if isinstance(value, Typed):
                value._name = name
                order.append(name)
        d['_order'] = order
        return type.__new__(cls, clsname, bases, d)

    @classmethod
    def __prepare__(cls, clsname, bases):
        return OrderedDict()
```

在这个元类中，执行类主体时描述器的定义顺序会被一个 `OrderedDict` 捕获到， 生成的有序名称从字典中提取出来并放入类属性 `_order` 中。这样的话类中的方法可以通过多种方式来使用它。 例如，下面是一个简单的类，使用这个排序字典来实现将一个类实例的数据序列化为一行CSV数据：

```
class Structure(metaclass=OrderedMeta):
    def as_csv(self):
        return ','.join(str(getattr(self,name)) for name in self._order)

# Example use
class Stock(Structure):
    name = String()
    shares = Integer()
    price = Float()

    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

我们在交互式环境中测试一下这个Stock类：

```
>>> s = Stock('GOOG',100,490.1)
>>> s.name
'GOOG'
>>> s.as_csv()
'GOOG,100,490.1'
>>> t = Stock('AAPL','a lot', 610.23)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "dupmethod.py", line 34, in __init__
TypeError: shares expects <class 'int'>
>>>
```

讨论¶

本节一个关键点就是OrderedMeta元类中定义的 `__prepare__()` 方法。 这个方法会在开始定义类和它的父类的时候被执行。它必须返回一个映射对象以便在类定义体中被使用到。 我们这里通过返回了一个OrderedDict而不是一个普通的字典，可以很容易的捕获定义的顺序。

如果你想构造自己的类字典对象，可以很容易的扩展这个功能。比如，下面的这个修改方案可以防止重复的定义：

```
from collections import OrderedDict

class NoDupOrderedDict(OrderedDict):
    def __init__(self, clsname):
        self.clsname = clsname
        super().__init__()
    def __setitem__(self, name, value):
        if name in self:
            raise TypeError('{} already defined in {}'.format(name, self.clsname))
        super().__setitem__(name, value)

class OrderedMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        d = dict(clsdict)
        d['_order'] = [name for name in clsdict if name[0] != '_']
        return type.__new__(cls, clsname, bases, d)

    @classmethod
    def __prepare__(cls, clsname, bases):
        return NoDupOrderedDict(clsname)
```

下面我们测试重复的定义会出现什么情况：

```
>>> class A(metaclass=OrderedMeta):
... def spam(self):
... pass
... def spam(self):
... pass
...
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "<stdin>", line 4, in A
    File "dupmethod2.py", line 25, in __setitem__
        (name, self.clsname))
TypeError: spam already defined in A
>>>
```

最后还有一点很重要，就是在 `__new__()` 方法中对于元类中被修改字典的处理。 尽管类使用了另外一个字典来定义，在构造最终的 `class` 对象的时候， 我们仍然需要将这个字典转换为一个正确的 `dict` 实例。 通过语句 `d = dict(clsdict)` 来完成这个效果。

对于很多应用程序而已，能够捕获类定义的顺序是一个看似不起眼却又非常重要的特性。 例如，在对象关系映射中，我们通常会看到下面这种方式定义的类：

```
class Stock(Model):
```

```
    name = String()
    shares = Integer()
    price = Float()
```

在框架底层，我们必须捕获定义的顺序来将对象映射到元组或数据库表中的行（就类似于上面例子中的 `as_csv()` 的功能）。 这节演示的技术非常简单，并且通常会比其他类似方法（通常都要在描述器类中维护一个隐藏的计数器）要简单的多。

9.15 定义有可选参数的元类¶

问题¶

你想定义一个元类，允许类定义时提供可选参数，这样可以控制或配置类型的创建过程。

解决方案¶

在定义类的时候，Python允许我们使用 ``metaclass``关键字参数来指定特定的元类。 例如使用抽象基类：

```
from abc import ABCMeta, abstractmethod
class IStream(metaclass=ABCMeta):
    @abstractmethod
    def read(self, maxsize=None):
        pass

    @abstractmethod
    def write(self, data):
        pass
```

然而，在自定义元类中我们还可以提供其他的关键字参数，如下所示：

```
class Spam(metaclass=MyMeta, debug=True, synchronize=True):
    pass
```

为了使元类支持这些关键字参数，你必须确保在 `__prepare__()`, `__new__()` 和 `__init__()` 方法中 都使用强制关键字参数。就像下面这样：

```
class MyMeta(type):
    # Optional
    @classmethod
    def __prepare__(cls, name, bases, *, debug=False, synchronize=False):
        # Custom processing
        pass
        return super().__prepare__(name, bases)

    # Required
    def __new__(cls, name, bases, ns, *, debug=False, synchronize=False):
        # Custom processing
        pass
        return super().__new__(cls, name, bases, ns)

    # Required
    def __init__(self, name, bases, ns, *, debug=False, synchronize=False):
        # Custom processing
        pass
        super().__init__(name, bases, ns)
```

讨论¶

给一个元类添加可选关键字参数需要你完全弄懂类创建的所有步骤， 因为这些参数会被传递给每一个相关的方法。 `__prepare__()` 方法在所有类定义开始执行前首先被调用，用来创建类命名空间。 通常来讲，这个方法只是简单的返回一个字典或其他映射对象。 `__new__()` 方法被用来实例化最终的类对象。它在类的主体被执行完后开始执行。 `__init__()` 方法最后被调用，用来执行其他的一些初始化工作。

当我们构造元类的时候，通常只需要定义一个 `__new__()` 或 `__init__()` 方法，但不是两个都定义。 但是，如果需要接受其他的关键字参数的话，这两个方法就要同时提供，并且都要提供对应的参数签名。 默认的 `__prepare__()` 方法接受任意的关键字参数，但是会忽略它们， 所以只有当这些额外的参数可能会影响到类命名空间的创建时你才需要去定义 `__prepare__()` 方法。

通过使用强制关键字参数，在类的创建过程中我们必须通过关键字来指定这些参数。

使用关键字参数配置一个元类还可以视作对类变量的一种替代方式。例如：

```
class Spam(metaclass=MyMeta):
    debug = True
    synchronize = True
    pass
```

将这些属性定义为参数的好处在于它们不会污染类的名称空间， 这些属性仅仅只从属于类的创建阶段，而不是类中的语句执行阶段。 另外，它们在 `__prepare__()` 方法中是可以被访问的，因为这个方法会在所有类主体执行前被执行。但是类变量只能在元类的 `__new__()` 和 `__init__()` 方法中可见。

9.16 *args和**kwargs的强制参数签名¶

问题¶

你有一个函数或方法，它使用*args和**kwargs作为参数，这样使得它比较通用， 但有时候你想检查传递进来的参数是不是某个你想要的类型。

解决方案¶

对任何涉及到操作函数调用签名的问题，你都应该使用 `inspect` 模块中的签名特性。 我们最主要关注两个类：`Signature` 和 `Parameter` 。下面是一个创建函数前面的交互例子：

```
>>> from inspect import Signature, Parameter
>>> # Make a signature for a func(x, y=42, *, z=None)
>>> parms = [ Parameter('x', Parameter.POSITIONAL_OR_KEYWORD),
...          Parameter('y', Parameter.POSITIONAL_OR_KEYWORD, default=42),
...          Parameter('z', Parameter.KEYWORD_ONLY, default=None) ]
>>> sig = Signature(parms)
>>> print(sig)
(x, y=42, *, z=None)
>>>
```

一旦你有了一个签名对象，你就可以使用它的 `bind()` 方法很容易的将它绑定到 `*args` 和 `**kwargs` 上去。 下面是一个简单的演示：

```
>>> def func(*args, **kwargs):
...     bound_values = sig.bind(*args, **kwargs)
...     for name, value in bound_values.arguments.items():
...         print(name,value)
...
>>> # Try various examples
>>> func(1, 2, z=3)
x 1
y 2
z 3
>>> func(1)
x 1
>>> func(1, z=3)
x 1
z 3
>>> func(y=2, x=1)
x 1
y 2
>>> func(1, 2, 3, 4)
Traceback (most recent call last):
...
    File "/usr/local/lib/python3.3/inspect.py", line 1972, in _bind
        raise TypeError('too many positional arguments')
TypeError: too many positional arguments
>>> func(y=2)
Traceback (most recent call last):
...
    File "/usr/local/lib/python3.3/inspect.py", line 1961, in _bind
        raise TypeError(msg) from None
TypeError: 'x' parameter lacking default value
>>> func(1, y=2, x=3)
Traceback (most recent call last):
...
    File "/usr/local/lib/python3.3/inspect.py", line 1985, in _bind
        '{arg!r}'.format(arg=param.name))
TypeError: multiple values for argument 'x'
>>>
```

可以看出来，通过将签名和传递的参数绑定起来，可以强制函数调用遵循特定的规则，比如必填、默认、重复等等。

下面是一个强制函数签名更具体的例子。在代码中，我们在基类中先定义了一个非常通用的 `__init__()` 方法， 然后我们强制所有的子类必须提供一个特定的参数签名。

```
from inspect import Signature, Parameter

def make_sig(*names):
    parms = [Parameter(name, Parameter.POSITIONAL_OR_KEYWORD)
             for name in names]
    return Signature(parms)

class Structure:
    __signature__ = make_sig()
    def __init__(self, *args, **kwargs):
        bound_values = self.__signature__.bind(*args, **kwargs)
        for name, value in bound_values.arguments.items():
            setattr(self, name, value)

# Example use
class Stock(Structure):
    __signature__ = make_sig('name', 'shares', 'price')

class Point(Structure):
    __signature__ = make_sig('x', 'y')
```

下面是使用这个 `Stock` 类的示例：

```
>>> import inspect
>>> print(inspect.signature(Stock))
(name, shares, price)
>>> s1 = Stock('ACME', 100, 490.1)
>>> s2 = Stock('ACME', 100)
Traceback (most recent call last):
...
TypeError: 'price' parameter lacking default value
>>> s3 = Stock('ACME', 100, 490.1, shares=50)
Traceback (most recent call last):
...
TypeError: multiple values for argument 'shares'
>>>
```

讨论¶

在我们需要构建通用函数库、编写装饰器或实现代理的时候，对于 `*args` 和 `**kwargs` 的使用是很普遍的。 但是，这样的函数有一个缺点就是当你想要实现自己的参数检验时，代码就会笨拙混乱。在8.11小节里面有这样一个例子。 这时候我们可以通过一个签名对象来简化它。

在最后的一个方案实例中，我们还可以通过使用自定义元类来创建签名对象。下面演示怎样来实现：

```
from inspect import Signature, Parameter

def make_sig(*names):
    parms = [Parameter(name, Parameter.POSITIONAL_OR_KEYWORD)
             for name in names]
    return Signature(parms)

class StructureMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        clsdict['__signature__'] = make_sig(*clsdict.get('_fields',[]))
        return super().__new__(cls, clsname, bases, clsdict)

class Structure(metaclass=StructureMeta):
    _fields = []
    def __init__(self, *args, **kwargs):
        bound_values = self.__signature__.bind(*args, **kwargs)
        for name, value in bound_values.arguments.items():
            setattr(self, name, value)

# Example
```

```
class Stock(Structure):
    _fields = ['name', 'shares', 'price']

class Point(Structure):
    _fields = ['x', 'y']
```

当我们自定义签名的时候，将签名存储在特定的属性 `__signature__` 中通常是很有用的。 这样的话，在使用 `inspect` 模块执行内省的代码就能发现签名并将它作为调用约定。

```
>>> import inspect
>>> print(inspect.signature(Stock))
(name, shares, price)
>>> print(inspect.signature(Point))
(x, y)
>>>
```

9.17 在类上强制使用编程规约¶

问题¶

你的程序包含一个很大的类继承体系，你希望强制执行某些编程规约（或者代码诊断）来帮助程序员保持清醒。

解决方案¶

如果你想监控类的定义，通常可以通过定义一个元类。一个基本元类通常是继承自 `type` 并重定义它的 `__new__()` 方法或者是 `__init__()` 方法。比如：

```
class MyMeta(type):
    def __new__(self, clsname, bases, clsdict):
        # clsname is name of class being defined
        # bases is tuple of base classes
        # clsdict is class dictionary
        return super().__new__(cls, clsname, bases, clsdict)
```

另一种是，定义 `__init__()` 方法：

```
class MyMeta(type):
    def __init__(self, clsname, bases, clsdict):
        super().__init__(clsname, bases, clsdict)
        # clsname is name of class being defined
        # bases is tuple of base classes
        # clsdict is class dictionary
```

为了使用这个元类，你通常要将它放到到一个顶级父类定义中，然后其他的类继承这个顶级父类。例如：

```
class Root(metaclass=MyMeta):
    pass

class A(Root):
    pass

class B(Root):
    pass
```

元类的一个关键特点是它允许你在定义的时候检查类的内容。在重新定义 `__init__()` 方法中， 你可以很轻松的检查类字典、父类等等。并且，一旦某个元类被指定给了某个类，那么就会被继承到所有子类中去。 因此，一个框架的构建者就能在大型的继承体系中通过给一个顶级父类指定一个元类去捕获所有下面子类的定义。

作为一个具体的应用例子，下面定义了一个元类，它会拒绝任何有混合大小写名字作为方法的类定义（可能是想气死Java程序员^_^）：

```
class NoMixedCaseMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        for name in clsdict:
            if name.lower() != name:
                raise TypeError('Bad attribute name: ' + name)
        return super().__new__(cls, clsname, bases, clsdict)

class Root(metaclass=NoMixedCaseMeta):
    pass

class A(Root):
    def foo_bar(self): # Ok
        pass

class B(Root):
    def fooBar(self): # TypeError
        pass
```

作为更高级和实用的例子，下面有一个元类，它用来检测重载方法，确保它的调用参数跟父类中原始方法有着相同的

参数签名。

```
from inspect import signature
import logging

class MatchSignaturesMeta(type):

    def __init__(self, clsname, bases, clsdict):
        super().__init__(clsname, bases, clsdict)
        sup = super(self, self)
        for name, value in clsdict.items():
            if name.startswith('_') or not callable(value):
                continue
            # Get the previous definition (if any) and compare the signatures
            prev_dfn = getattr(sup,name,None)
            if prev_dfn:
                prev_sig = signature(prev_dfn)
                val_sig = signature(value)
                if prev_sig != val_sig:
                    logging.warning('Signature mismatch in %s. %s != %s',
                                    value.__qualname__, prev_sig, val_sig)

# Example
class Root(metaclass=MatchSignaturesMeta):
    pass

class A(Root):
    def foo(self, x, y):
        pass

    def spam(self, x, *, z):
        pass

# Class with redefined methods, but slightly different signatures
class B(A):
    def foo(self, a, b):
        pass

    def spam(self,x,z):
        pass
```

如果你运行这段代码，就会得到下面这样的输出结果：

```
WARNING:root:Signature mismatch in B.spam. (self, x, *, z) != (self, x, z)
WARNING:root:Signature mismatch in B.foo. (self, x, y) != (self, a, b)
```

这种警告信息对于捕获一些微妙的程序bug是很有用的。例如，如果某个代码依赖于传递给方法的关键字参数， 那么当子类改变参数名字的时候就会调用出错。

讨论¶

在大型面向对象的程序中，通常将类的定义放在元类中控制是很有用的。 元类可以监控类的定义，警告编程人员某些没有注意到的可能出现的问题。

有人可能会说，像这样的错误可以通过程序分析工具或IDE去做会更好些。诚然，这些工具是很有用。 但是，如果你在构建一个框架或函数库供其他人使用，那么你没办法去控制使用者要使用什么工具。 因此，对于这种类型的程序，如果可以在元类中做检测或许可以带来更好的用户体验。

在元类中选择重新定义 `__new__()` 方法还是 `__init__()` 方法取决于你想怎样使用结果类。 `__new__()` 方法在类创建之前被调用，通常用于通过某种方式（比如通过改变类字典的内容）修改类的定义。 而 `__init__()` 方法是在类被创建之后被调用，当你需要完整构建类对象的时候会很有用。 在最后一个例子中，这是必要的，因为它使用了 `super()` 函数来搜索之前的定义。 它只能在类的实例被创建之后，并且相应的方法解析顺序也已经被设置好了。

最后一个例子还演示了Python的函数签名对象的使用。 实际上，元类将每个可调用定义放在一个类中，搜索前一个定义（如果有的话）， 然后通过使用 `inspect.signature()` 来简单的比较它们的调用签名。

最后一点，代码中有一行使用了 `super(self, self)` 并不是排版错误。 当使用元类的时候，我们要时刻记住一点就是 `self` 实际上是一个类对象。 因此，这条语句其实就是用来寻找位于继承体系中构建 `self` 父类的定义。

9.18 以编程方式定义类¶

问题¶

你在写一段代码，最终需要创建一个新的类对象。你考虑将类的定义源代码以字符串的形式发布出去。 并且使用函数比如 `exec()` 来执行它，但是你想寻找一个更加优雅的解决方案。

解决方案¶

你可以使用函数 `types.new_class()` 来初始化新的类对象。 你需要做的只是提供类的名字、父类元组、关键字参数，以及一个用成员变量填充类字典的回调函数。例如：

```
# stock.py
# Example of making a class manually from parts

# Methods
def __init__(self, name, shares, price):
    self.name = name
    self.shares = shares
    self.price = price
def cost(self):
    return self.shares * self.price

cls_dict = {
    '__init__' : __init__,
    'cost' : cost,
}

# Make a class
import types

Stock = types.new_class('Stock', (), {}, lambda ns: ns.update(cls_dict))
Stock.__module__ = __name__
```

这种方式会构建一个普通的类对象，并且按照你的期望工作：

```
>>> s = Stock('ACME', 50, 91.1)
>>> s
<stock.Stock object at 0x1006a9b10>
>>> s.cost()
4555.0
>>>
```

这种方法中，一个比较难理解的地方是在调用完 `types.new_class()` 对 `Stock.__module__` 的赋值。 每次当一个类被定义后，它的 `__module__` 属性包含定义它的模块名。 这个名字用于生成 `__repr__()` 方法的输出。它同样也被用于很多库，比如 `pickle` 。 因此，为了让你创建的类是“正确”的，你需要确保这个属性也设置正确了。

如果你想创建的类需要一个不同的元类，可以通过 `types.new_class()` 第三个参数传递给它。例如：

```
>>> import abc
>>> Stock = types.new_class('Stock', (), {'metaclass': abc.ABCMeta},
...                         lambda ns: ns.update(cls_dict))
...
>>> Stock.__module__ = __name__
>>> Stock
<class '__main__.Stock'>
>>> type(Stock)
<class 'abc.ABCMeta'>
>>>
```

第三个参数还可以包含其他的关键字参数。比如，一个类的定义如下：

```
class Spam(Base, debug=True, typecheck=False):
    pass
```

那么可以将其翻译成如下的 `new_class()` 调用形式：

```
Spam = types.new_class('Spam', (Base,),
                       {'debug': True, 'typecheck': False},
                       lambda ns: ns.update(cls_dict))
```

`new_class()` 第四个参数最神秘，它是一个用来接受类命名空间的映射对象的函数。 通常这是一个普通的字典，但是它实际上是 `__prepare__()` 方法返回的任意对象，这个在9.14小节已经介绍过了。 这个函数需要使用上面演示的 `update()` 方法给命名空间增加内容。

讨论¶

很多时候如果能构造新的类对象是很有用的。 有个很熟悉的例子是调用 `collections.namedtuple()` 函数，例如：

```
>>> Stock = collections.namedtuple('Stock', ['name', 'shares', 'price'])
>>> Stock
<class '__main__.Stock'>
>>>
```

`namedtuple()` 使用 `exec()` 而不是上面介绍的技术。但是，下面通过一个简单的变化， 我们直接创建一个类：

```
import operator
import types
import sys

def named_tuple(classname, fieldnames):
    # Populate a dictionary of field property accessors
    cls_dict = { name: property(operator.itemgetter(n))
                 for n, name in enumerate(fieldnames) }

    # Make a __new__ function and add to the class dict
    def __new__(cls, *args):
        if len(args) != len(fieldnames):
            raise TypeError('Expected {} arguments'.format(len(fieldnames)))
        return tuple.__new__(cls, args)

    cls_dict['__new__'] = __new__

    # Make the class
    cls = types.new_class(classname, (tuple,), {},
                          lambda ns: ns.update(cls_dict))

    # Set the module to that of the caller
    cls.__module__ = sys._getframe(1).f_globals['__name__']
    return cls
```

这段代码的最后部分使用了一个所谓的”框架魔法”，通过调用 `sys._getframe()` 来获取调用者的模块名。 另外一个框架魔法例子在2.15小节中有介绍过。

下面的例子演示了前面的代码是如何工作的：

```
>>> Point = named_tuple('Point', ['x', 'y'])
>>> Point
<class '__main__.Point'>
>>> p = Point(4, 5)
>>> len(p)
2
>>> p.x
4
>>> p.y
5
>>> p.x = 2
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>> print('%s %s' % p)
4 5
```

```
>>>
```

这项技术一个很重要的方面是它对于元类的正确使用。 你可能像通过直接实例化一个元类来直接创建一个类：

```
Stock = type('Stock', (), cls_dict)
```

这种方法的问题在于它忽略了一些关键步骤，比如对于元类中 `__prepare__()` 方法的调用。 通过使用 `types.new_class()` ，你可以保证所有的必要初始化步骤都能得到执行。 比如，`types.new_class()` 第四个参数的回调函数接受 `__prepare__()` 方法返回的映射对象。

如果你仅仅只是想执行准备步骤，可以使用 `types.prepare_class()` 。例如：

```
import types
metaclass, kwargs, ns = types.prepare_class('Stock', (), {'metaclass': type})
```

它会查找合适的元类并调用它的 `__prepare__()` 方法。 然后这个元类保存它的关键字参数，准备命名空间后被返回。

更多信息, 请参考 PEP 3115 , 以及 Python documentation .

9.19 在定义的时候初始化类的成员¶

问题¶

你想在类被定义的时候就初始化一部分类的成员，而不是要等到实例被创建后。

解决方案¶

在类定义时就执行初始化或设置操作是元类的一个典型应用场景。本质上讲，一个元类会在定义时被触发， 这时候你可以执行一些额外的操作。

下面是一个例子，利用这个思路来创建类似于 `collections` 模块中的命名元组的类：

```
import operator

class StructTupleMeta(type):
    def __init__(cls, *args, **kwargs):
        super().__init__(*args, **kwargs)
        for n, name in enumerate(cls._fields):
            setattr(cls, name, property(operator.itemgetter(n)))

class StructTuple(tuple, metaclass=StructTupleMeta):
    _fields = []
    def __new__(cls, *args):
        if len(args) != len(cls._fields):
            raise ValueError('{} arguments required'.format(len(cls._fields)))
        return super().__new__(cls,args)
```

这段代码可以用来定义简单的基于元组的数据结构，如下所示：

```
class Stock(StructTuple):
    _fields = ['name', 'shares', 'price']

class Point(StructTuple):
    _fields = ['x', 'y']
```

下面演示它如何工作：

```
>>> s = Stock('ACME', 50, 91.1)
>>> s
('ACME', 50, 91.1)
>>> s[0]
'ACME'
>>> s.name
'ACME'
>>> s.shares * s.price
4555.0
>>> s.shares = 23
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

讨论¶

这一小节中，类 `StructTupleMeta` 获取到类属性 `_fields` 中的属性名字列表， 然后将它们转换成相应的可访问特定元组槽的方法。函数 `operator.itemgetter()` 创建一个访问器函数， 然后 `property()` 函数将其转换成一个属性。

本节最难懂的部分是知道不同的初始化步骤是什么时候发生的。 `StructTupleMeta` 中的 `__init__()` 方法只在每个类被定义时被调用一次。 `cls` 参数就是那个被定义的类。实际上，上述代码使用了 `_fields` 类变量来保存新的被定义的类， 然后给它再添加一点新的东西。

`StructTuple` 类作为一个普通的基类，供其他使用者来继承。 这个类中的 `__new__()` 方法用来构造新的实例。 这里使

用 `__new__()` 并不是很常见，主要是因为我们要修改元组的调用签名， 使得我们可以像普通的实例调用那样创建实例。就像下面这样：

```
s = Stock('ACME', 50, 91.1) # OK
s = Stock(('ACME', 50, 91.1)) # Error
```

跟 `__init__()` 不同的是，`__new__()` 方法在实例被创建之前被触发。 由于元组是不可修改的，所以一旦它们被创建了就不可能对它做任何改变。而 `__init__()` 会在实例创建的最后被触发， 这样的话我们就可以做我们想做的了。这也是为什么 `__new__()` 方法已经被定义了。

尽管本节很短，还是需要你能仔细研读，深入思考Python类是如何被定义的，实例是如何被创建的， 还有就是元类和类的各个不同的方法究竟在什么时候被调用。

PEP 422 提供了一个解决本节问题的另外一种方法。 但是，截止到我写这本书的时候，它还没被采纳和接受。 尽管如此，如果你使用的是Python 3.3或更高的版本，那么还是值得去看一下的。

9.2 创建装饰器时保留函数元信息¶

问题¶

你写了一个装饰器作用在某个函数上，但是这个函数的重要的元信息比如名字、文档字符串、注解和参数签名都丢失了。

解决方案¶

任何时候你定义装饰器的时候，都应该使用 `functools` 库中的 `@wraps` 装饰器来注解底层包装函数。例如：

```
import time
from functools import wraps
def timethis(func):
    '''
    Decorator that reports the execution time.
    '''
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end-start)
        return result
    return wrapper
```

下面我们使用这个被包装后的函数并检查它的元信息：

```
>>> @timethis
... def countdown(n):
...     '''
...     Counts down
...     '''
...     while n > 0:
...         n -= 1
...
>>> countdown(100000)
countdown 0.008917808532714844
>>> countdown.__name__
'countdown'
>>> countdown.__doc__
'\n\tCounts down\n\t'
>>> countdown.__annotations__
{'n': <class 'int'>}
>>>
```

讨论¶

在编写装饰器的时候复制元信息是一个非常重要的部分。如果你忘记了使用 `@wraps`， 那么你会发现被装饰函数丢失了所有有用的信息。比如如果忽略 `@wraps` 后的效果是下面这样的：

```
>>> countdown.__name__
'wrapper'
>>> countdown.__doc__
>>> countdown.__annotations__
{}
>>>
```

`@wraps` 有一个重要特征是它能让你通过属性 `__wrapped__` 直接访问被包装函数。例如:

```
>>> countdown.__wrapped__(100000)
>>>
```

`__wrapped__` 属性还能让被装饰函数正确暴露底层的参数签名信息。例如：

```
>>> from inspect import signature
>>> print(signature(countdown))
(n:int)
>>>
```

一个很普遍的问题是怎样让装饰器去直接复制原始函数的参数签名信息， 如果想自己手动实现的话需要做大量的工作，最好就简单的使用 `@wraps` 装饰器。 通过底层的 `__wrapped__` 属性访问到函数签名信息。更多关于签名的内容可以参考9.16小节。

9.20 利用函数注解实现方法重载¶

问题¶

你已经学过怎样使用函数参数注解，那么你可能会想利用它来实现基于类型的方法重载。 但是你不确定应该怎样去实现（或者到底行得通不）。

解决方案¶

本小节的技术是基于一个简单的技术，那就是Python允许参数注解，代码可以像下面这样写：

```
class Spam:
    def bar(self, x:int, y:int):
        print('Bar 1:', x, y)

    def bar(self, s:str, n:int = 0):
        print('Bar 2:', s, n)

s = Spam()
s.bar(2, 3) # Prints Bar 1: 2 3
s.bar('hello') # Prints Bar 2: hello 0
```

下面是我们第一步的尝试，使用到了一个元类和描述器：

```
# multiple.py
import inspect
import types

class MultiMethod:
    '''
    Represents a single multimethod.
    '''
    def __init__(self, name):
        self._methods = {}
        self.__name__ = name

    def register(self, meth):
        '''
        Register a new method as a multimethod
        '''
        sig = inspect.signature(meth)

        # Build a type signature from the method's annotations
        types = []
        for name, parm in sig.parameters.items():
            if name == 'self':
                continue
            if parm.annotation is inspect.Parameter.empty:
                raise TypeError(
                    'Argument {} must be annotated with a type'.format(name)
                )
            if not isinstance(parm.annotation, type):
                raise TypeError(
                    'Argument {} annotation must be a type'.format(name)
                )
            if parm.default is not inspect.Parameter.empty:
                self._methods[tuple(types)] = meth
            types.append(parm.annotation)

        self._methods[tuple(types)] = meth

    def __call__(self, *args):
        '''
        Call a method based on type signature of the arguments
        '''
        types = tuple(type(arg) for arg in args[1:])
```

```
        meth = self._methods.get(types, None)
        if meth:
            return meth(*args)
        else:
            raise TypeError('No matching method for types {}'.format(types))

    def __get__(self, instance, cls):
        '''
        Descriptor method needed to make calls work in a class
        '''
        if instance is not None:
            return types.MethodType(self, instance)
        else:
            return self

class MultiDict(dict):
    '''
    Special dictionary to build multimethods in a metaclass
    '''
    def __setitem__(self, key, value):
        if key in self:
            # If key already exists, it must be a multimethod or callable
            current_value = self[key]
            if isinstance(current_value, MultiMethod):
                current_value.register(value)
            else:
                mvalue = MultiMethod(key)
                mvalue.register(current_value)
                mvalue.register(value)
                super().__setitem__(key, mvalue)
        else:
            super().__setitem__(key, value)

class MultipleMeta(type):
    '''
    Metaclass that allows multiple dispatch of methods
    '''
    def __new__(cls, clsname, bases, clsdict):
        return type.__new__(cls, clsname, bases, dict(clsdict))

    @classmethod
    def __prepare__(cls, clsname, bases):
        return MultiDict()
```

为了使用这个类，你可以像下面这样写：

```
class Spam(metaclass=MultipleMeta):
    def bar(self, x:int, y:int):
        print('Bar 1:', x, y)

    def bar(self, s:str, n:int = 0):
        print('Bar 2:', s, n)

# Example: overloaded __init__
import time

class Date(metaclass=MultipleMeta):
    def __init__(self, year: int, month:int, day:int):
        self.year = year
        self.month = month
        self.day = day

    def __init__(self):
        t = time.localtime()
        self.__init__(t.tm_year, t.tm_mon, t.tm_mday)
```

下面是一个交互示例来验证它能正确的工作：

```
>>> s = Spam()
>>> s.bar(2, 3)
```

```
Bar 1: 2 3
>>> s.bar('hello')
Bar 2: hello 0
>>> s.bar('hello', 5)
Bar 2: hello 5
>>> s.bar(2, 'hello')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "multiple.py", line 42, in __call__
        raise TypeError('No matching method for types {}'.format(types))
TypeError: No matching method for types (<class 'int'>, <class 'str'>)
>>> # Overloaded __init__
>>> d = Date(2012, 12, 21)
>>> # Get today's date
>>> e = Date()
>>> e.year
2012
>>> e.month
12
>>> e.day
3
>>>
```

讨论¶

坦白来讲，相对于通常的代码而已本节使用到了很多的魔法代码。 但是，它却能让我们深入理解元类和描述器的底层工作原理， 并能加深对这些概念的印象。因此，就算你并不会立即去应用本节的技术， 它的一些底层思想却会影响到其它涉及到元类、描述器和函数注解的编程技术。

本节的实现中的主要思路其实是很简单的。`MultipleMeta` 元类使用它的 `__prepare__()` 方法 来提供一个作为 `MultiDict` 实例的自定义字典。这个跟普通字典不一样的是， `MultiDict` 会在元素被设置的时候检查是否已经存在，如果存在的话，重复的元素会在 `MultiMethod` 实例中合并。

`MultiMethod` 实例通过构建从类型签名到函数的映射来收集方法。 在这个构建过程中，函数注解被用来收集这些签名然后构建这个映射。 这个过程在 `MultiMethod.register()` 方法中实现。 这种映射的一个关键特点是对于多个方法，所有参数类型都必须要指定，否则就会报错。

为了让 `MultiMethod` 实例模拟一个调用，它的 `__call__()` 方法被实现了。 这个方法从所有排除 `self` 的参数中构建一个类型元组，在内部map中查找这个方法， 然后调用相应的方法。为了能让 `MultiMethod` 实例在类定义时正确操作， `__get__()` 是必须得实现的。 它被用来构建正确的绑定方法。比如：

```
>>> b = s.bar
>>> b
<bound method Spam.bar of <__main__.Spam object at 0x1006a46d0>>
>>> b.__self__
<__main__.Spam object at 0x1006a46d0>
>>> b.__func__
<__main__.MultiMethod object at 0x1006a4d50>
>>> b(2, 3)
Bar 1: 2 3
>>> b('hello')
Bar 2: hello 0
>>>
```

不过本节的实现还有一些限制，其中一个是它不能使用关键字参数。例如：

```
>>> s.bar(x=2, y=3)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: __call__() got an unexpected keyword argument 'y'

>>> s.bar(s='hello')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: __call__() got an unexpected keyword argument 's'
>>>
```

也许有其他的方法能添加这种支持，但是它需要一个完全不同的方法映射方式。 问题在于关键字参数的出现是没有顺序的。当它跟位置参数混合使用时， 那你的参数就会变得比较混乱了，这时候你不得不在 `__call__()` 方法中先去做个排序。

同样对于继承也是有限制的，例如，类似下面这种代码就不能正常工作：

```
class A:
    pass

class B(A):
    pass

class C:
    pass

class Spam(metaclass=MultipleMeta):
    def foo(self, x:A):
        print('Foo 1:', x)

    def foo(self, x:C):
        print('Foo 2:', x)
```

原因是因为 `x:A` 注解不能成功匹配子类实例（比如B的实例），如下：

```
>>> s = Spam()
>>> a = A()
>>> s.foo(a)
Foo 1: <__main__.A object at 0x1006a5310>
>>> c = C()
>>> s.foo(c)
Foo 2: <__main__.C object at 0x1007a1910>
>>> b = B()
>>> s.foo(b)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "multiple.py", line 44, in __call__
        raise TypeError('No matching method for types {}'.format(types))
TypeError: No matching method for types (<class '__main__.B'>,)
>>>
```

作为使用元类和注解的一种替代方案，可以通过描述器来实现类似的效果。例如：

```
import types

class multimethod:
    def __init__(self, func):
        self._methods = {}
        self.__name__ = func.__name__
        self._default = func

    def match(self, *types):
        def register(func):
            ndefaults = len(func.__defaults__) if func.__defaults__ else 0
            for n in range(ndefaults+1):
                self._methods[types[:len(types) - n]] = func
            return self
        return register

    def __call__(self, *args):
        types = tuple(type(arg) for arg in args[1:])
        meth = self._methods.get(types, None)
        if meth:
            return meth(*args)
        else:
            return self._default(*args)

    def __get__(self, instance, cls):
        if instance is not None:
            return types.MethodType(self, instance)
```

```
        else:
            return self
```

为了使用描述器版本，你需要像下面这样写：

```
class Spam:
    @multimethod
    def bar(self, *args):
        # Default method called if no match
        raise TypeError('No matching method for bar')

    @bar.match(int, int)
    def bar(self, x, y):
        print('Bar 1:', x, y)

    @bar.match(str, int)
    def bar(self, s, n = 0):
        print('Bar 2:', s, n)
```

描述器方案同样也有前面提到的限制（不支持关键字参数和继承）。

所有事物都是平等的，有好有坏，也许最好的办法就是在普通代码中避免使用方法重载。 不过有些特殊情况下还是有意义的，比如基于模式匹配的方法重载程序中。 举个例子，8.21小节中的访问者模式可以修改为一个使用方法重载的类。 但是，除了这个以外，通常不应该使用方法重载（就简单的使用不同名称的方法就行了）。

在Python社区对于实现方法重载的讨论已经由来已久。 对于引发这个争论的原因，可以参考下Guido van Rossum的这篇博客： Five-Minute Multimethods in Python

9.21 避免重复的属性方法¶

问题¶

你在类中需要重复的定义一些执行相同逻辑的属性方法，比如进行类型检查，怎样去简化这些重复代码呢？

解决方案¶

考虑下一个简单的类，它的属性由属性方法包装：

```
class Person:
    def __init__(self, name ,age):
        self.name = name
        self.age = age

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        if not isinstance(value, str):
            raise TypeError('name must be a string')
        self._name = value

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if not isinstance(value, int):
            raise TypeError('age must be an int')
        self._age = value
```

可以看到，为了实现属性值的类型检查我们写了很多的重复代码。 只要你以后看到类似这样的代码，你都应该想办法去简化它。 一个可行的方法是创建一个函数用来定义属性并返回它。例如：

```
def typed_property(name, expected_type):
    storage_name = '_' + name

    @property
    def prop(self):
        return getattr(self, storage_name)

    @prop.setter
    def prop(self, value):
        if not isinstance(value, expected_type):
            raise TypeError('{} must be a {}'.format(name, expected_type))
        setattr(self, storage_name, value)

    return prop

# Example use
class Person:
    name = typed_property('name', str)
    age = typed_property('age', int)

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

讨论¶

本节我们演示内部函数或者闭包的一个重要特性，它们很像一个宏。例子中的函数 `typed_property()` 看上去有点难理

解，其实它所做的仅仅就是为你生成属性并返回这个属性对象。 因此，当在一个类中使用它的时候，效果跟将它里面的代码放到类定义中去是一样的。 尽管属性的 `getter` 和 `setter` 方法访问了本地变量如 `name`，`expected_type` 以及 `storate_name`，这个很正常，这些变量的值会保存在闭包当中。

我们还可以使用 `functools.partial()` 来稍稍改变下这个例子，很有趣。例如，你可以像下面这样：

```
from functools import partial

String = partial(typed_property, expected_type=str)
Integer = partial(typed_property, expected_type=int)

# Example:
class Person:
    name = String('name')
    age = Integer('age')

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

其实你可以发现，这里的代码跟8.13小节中的类型系统描述器代码有些相似。

9.22 定义上下文管理器的简单方法¶

问题¶

你想自己去实现一个新的上下文管理器，以便使用with语句。

解决方案¶

实现一个新的上下文管理器的最简单的方法就是使用 `contexlib` 模块中的 `@contextmanager` 装饰器。 下面是一个实现了代码块计时功能的上下文管理器例子：

```
import time
from contextlib import contextmanager

@contextmanager
def timethis(label):
    start = time.time()
    try:
        yield
    finally:
        end = time.time()
        print('{}: {}'.format(label, end - start))

# Example use
with timethis('counting'):
    n = 10000000
    while n > 0:
        n -= 1
```

在函数 `timethis()` 中，`yield` 之前的代码会在上下文管理器中作为 `__enter__()` 方法执行， 所有在 `yield` 之后的代码会作为 `__exit__()` 方法执行。 如果出现了异常，异常会在yield语句那里抛出。

下面是一个更加高级一点的上下文管理器，实现了列表对象上的某种事务：

```
@contextmanager
def list_transaction(orig_list):
    working = list(orig_list)
    yield working
    orig_list[:] = working
```

这段代码的作用是任何对列表的修改只有当所有代码运行完成并且不出现异常的情况下才会生效。 下面我们来演示一下：

```
>>> items = [1, 2, 3]
>>> with list_transaction(items) as working:
...     working.append(4)
...     working.append(5)
...
>>> items
[1, 2, 3, 4, 5]
>>> with list_transaction(items) as working:
...     working.append(6)
...     working.append(7)
...     raise RuntimeError('oops')
...
Traceback (most recent call last):
    File "<stdin>", line 4, in <module>
RuntimeError: oops
>>> items
[1, 2, 3, 4, 5]
>>>
```

讨论¶

通常情况下，如果要写一个上下文管理器，你需要定义一个类，里面包含一个 `__enter__()` 和一个 `__exit__()` 方法，如下所示：

```
import time

class timethis:
    def __init__(self, label):
        self.label = label

    def __enter__(self):
        self.start = time.time()

    def __exit__(self, exc_ty, exc_val, exc_tb):
        end = time.time()
        print('{}: {}'.format(self.label, end - self.start))
```

尽管这个也不难写，但是相比较写一个简单的使用 `@contextmanager` 注解的函数而言还是稍显乏味。

`@contextmanager` 应该仅仅用来写自包含的上下文管理函数。 如果你有一些对象(比如一个文件、网络连接或锁)，需要支持 `with` 语句，那么你就需要单独实现 `__enter__()` 方法和 `__exit__()` 方法。

9.23 在局部变量域中执行代码¶

问题¶

你想在使用范围内执行某个代码片段，并且希望在执行后所有的结果都不可见。

解决方案¶

为了理解这个问题，先试试一个简单场景。首先，在全局命名空间内执行一个代码片段：

```
>>> a = 13
>>> exec('b = a + 1')
>>> print(b)
14
>>>
```

然后，再在一个函数中执行同样的代码：

```
>>> def test():
...     a = 13
...     exec('b = a + 1')
...     print(b)
...
>>> test()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "<stdin>", line 4, in test
NameError: global name 'b' is not defined
>>>
```

可以看出，最后抛出了一个NameError异常，就跟在 `exec()` 语句从没执行过一样。 要是你想在后面的计算中使用到 `exec()` 执行结果的话就会有问题了。

为了修正这样的错误，你需要在调用 `exec()` 之前使用 `locals()` 函数来得到一个局部变量字典。 之后你就能从局部字典中获取修改过后的变量值了。例如：

```
>>> def test():
...     a = 13
...     loc = locals()
...     exec('b = a + 1')
...     b = loc['b']
...     print(b)
...
>>> test()
14
>>>
```

讨论¶

实际上对于 `exec()` 的正确使用是比较难的。大多数情况下当你要考虑使用 `exec()` 的时候， 还有另外更好的解决方案（比如装饰器、闭包、元类等等）。

然而，如果你仍然要使用 `exec()` ，本节列出了一些如何正确使用它的方法。 默认情况下，`exec()` 会在调用者局部和全局范围内执行代码。然而，在函数里面， 传递给 `exec()` 的局部范围是拷贝实际局部变量组成的一个字典。 因此，如果 `exec()` 如果执行了修改操作，这种修改后的结果对实际局部变量值是没有影响的。 下面是另外一个演示它的例子：

```
>>> def test1():
...     x = 0
...     exec('x += 1')
...     print(x)
...
>>> test1()
```

```
0
>>>
```

上面代码里，当你调用 `locals()` 获取局部变量时，你获得的是传递给 `exec()` 的局部变量的一个拷贝。 通过在代码执行后审查这个字典的值，那就能获取修改后的值了。下面是一个演示例子：

```
>>> def test2():
...     x = 0
...     loc = locals()
...     print('before:', loc)
...     exec('x += 1')
...     print('after:', loc)
...     print('x =', x)
...
>>> test2()
before: {'x': 0}
after: {'loc': {...}, 'x': 1}
x = 0
>>>
```

仔细观察最后一步的输出，除非你将 `loc` 中被修改后的值手动赋值给x，否则x变量值是不会变的。

在使用 `locals()` 的时候，你需要注意操作顺序。每次它被调用的时候， `locals()` 会获取局部变量值中的值并覆盖字典中相应的变量。 请注意观察下下面这个试验的输出结果：

```
>>> def test3():
...     x = 0
...     loc = locals()
...     print(loc)
...     exec('x += 1')
...     print(loc)
...     locals()
...     print(loc)
...
>>> test3()
{'x': 0}
{'loc': {...}, 'x': 1}
{'loc': {...}, 'x': 0}
>>>
```

注意最后一次调用 `locals()` 的时候x的值是如何被覆盖掉的。

作为 `locals()` 的一个替代方案，你可以使用你自己的字典，并将它传递给 `exec()` 。例如：

```
>>> def test4():
...     a = 13
...     loc = { 'a' : a }
...     glb = { }
...     exec('b = a + 1', glb, loc)
...     b = loc['b']
...     print(b)
...
>>> test4()
14
>>>
```

大部分情况下，这种方式是使用 `exec()` 的最佳实践。 你只需要保证全局和局部字典在后面代码访问时已经被初始化。

还有一点，在使用 `exec()` 之前，你可能需要问下自己是否有其他更好的替代方案。 大多数情况下当你要考虑使用 `exec()` 的时候， 还有另外更好的解决方案，比如装饰器、闭包、元类，或其他一些元编程特性。

9.24 解析与分析Python源码¶

问题¶

你想写解析并分析Python源代码的程序。

解决方案¶

大部分程序员知道Python能够计算或执行字符串形式的源代码。例如：

```
>>> x = 42
>>> eval('2 + 3*4 + x')
56
>>> exec('for i in range(10): print(i)')
0
1
2
3
4
5
6
7
8
9
>>>
```

尽管如此，`ast` 模块能被用来将Python源码编译成一个可被分析的抽象语法树（AST）。例如：

```
>>> import ast
>>> ex = ast.parse('2 + 3*4 + x', mode='eval')
>>> ex
<_ast.Expression object at 0x1007473d0>
>>> ast.dump(ex)
"Expression(body=BinOp(left=BinOp(left=Num(n=2), op=Add(),
right=BinOp(left=Num(n=3), op=Mult(), right=Num(n=4))), op=Add(),
right=Name(id='x', ctx=Load())))"

>>> top = ast.parse('for i in range(10): print(i)', mode='exec')
>>> top
<_ast.Module object at 0x100747390>
>>> ast.dump(top)
"Module(body=[For(target=Name(id='i', ctx=Store()),
iter=Call(func=Name(id='range', ctx=Load()), args=[Num(n=10)],
keywords=[], starargs=None, kwargs=None),
body=[Expr(value=Call(func=Name(id='print', ctx=Load()),
args=[Name(id='i', ctx=Load())], keywords=[], starargs=None,
kwargs=None))], orelse=[])])"
>>>
```

分析源码树需要你自己更多的学习，它是由一系列AST节点组成的。 分析这些节点最简单的方法就是定义一个访问者类，实现很多 `visit_NodeName()` 方法， `NodeName()` 匹配那些你感兴趣的节点。下面是这样一个类，记录了哪些名字被加载、存储和删除的信息。

```
import ast

class CodeAnalyzer(ast.NodeVisitor):
    def __init__(self):
        self.loaded = set()
        self.stored = set()
        self.deleted = set()

    def visit_Name(self, node):
        if isinstance(node.ctx, ast.Load):
            self.loaded.add(node.id)
        elif isinstance(node.ctx, ast.Store):
            self.stored.add(node.id)
```

```
        elif isinstance(node.ctx, ast.Del):
            self.deleted.add(node.id)

# Sample usage
if __name__ == '__main__':
    # Some Python code
    code = '''
    for i in range(10):
        print(i)
    del i
    '''

    # Parse into an AST
    top = ast.parse(code, mode='exec')

    # Feed the AST to analyze name usage
    c = CodeAnalyzer()
    c.visit(top)
    print('Loaded:', c.loaded)
    print('Stored:', c.stored)
    print('Deleted:', c.deleted)
```

如果你运行这个程序，你会得到下面这样的输出：

```
Loaded: {'i', 'range', 'print'}
Stored: {'i'}
Deleted: {'i'}
```

最后，AST可以通过 `compile()` 函数来编译并执行。例如：

```
>>> exec(compile(top,'<stdin>', 'exec'))
0
1
2
3
4
5
6
7
8
9
>>>
```

讨论¶

当你能够分析源代码并从中获取信息的时候，你就能写很多代码分析、优化或验证工具了。 例如，相比盲目的传递一些代码片段到类似 `exec()` 函数中，你可以先将它转换成一个AST， 然后观察它的细节看它到底是怎样做的。 你还可以写一些工具来查看某个模块的全部源码，并且在此基础上执行某些静态分析。

需要注意的是，如果你知道自己在干啥，你还能够重写AST来表示新的代码。 下面是一个装饰器例子，可以通过重新解析函数体源码、 重写AST并重新创建函数代码对象来将全局访问变量降为函数体作用范围，

```
# namelower.py
import ast
import inspect

# Node visitor that lowers globally accessed names into
# the function body as local variables.
class NameLower(ast.NodeVisitor):
    def __init__(self, lowered_names):
        self.lowered_names = lowered_names

    def visit_FunctionDef(self, node):
        # Compile some assignments to lower the constants
        code = '__globals = globals()\n'
        code += '\n'.join("{0} = __globals['{0}']".format(name)
                          for name in self.lowered_names)
        code_ast = ast.parse(code, mode='exec')
```

```
        # Inject new statements into the function body
        node.body[:0] = code_ast.body

        # Save the function object
        self.func = node

# Decorator that turns global names into locals
def lower_names(*namelist):
    def lower(func):
        srclines = inspect.getsource(func).splitlines()
        # Skip source lines prior to the @lower_names decorator
        for n, line in enumerate(srclines):
            if '@lower_names' in line:
                break

        src = '\n'.join(srclines[n+1:])
        # Hack to deal with indented code
        if src.startswith((' ','\t')):
            src = 'if 1:\n' + src
        top = ast.parse(src, mode='exec')

        # Transform the AST
        cl = NameLower(namelist)
        cl.visit(top)

        # Execute the modified AST
        temp = {}
        exec(compile(top,'','exec'), temp, temp)

        # Pull out the modified code object
        func.__code__ = temp[func.__name__].__code__
        return func
    return lower
```

为了使用这个代码，你可以像下面这样写：

```
INCR = 1
@lower_names('INCR')
def countdown(n):
    while n > 0:
        n -= INCR
```

装饰器会将 `countdown()` 函数重写为类似下面这样子：

```
def countdown(n):
    __globals = globals()
    INCR = __globals['INCR']
    while n > 0:
        n -= INCR
```

在性能测试中，它会让函数运行快20%

现在，你是不是想为你所有的函数都加上这个装饰器呢？或许不会。 但是，这却是对于一些高级技术比如AST操作、源码操作等等的一个很好的演示说明

本节受另外一个在 `ActiveState` 中处理Python字节码的章节的启示。 使用AST是一个更加高级点的技术，并且也更简单些。参考下面一节获得字节码的更多信息。

9.25 拆解Python字节码¶

问题¶

你想通过将你的代码反编译成低级的字节码来查看它底层的工作机制。

解决方案¶

`dis` 模块可以被用来输出任何Python函数的反编译结果。例如：

```
>>> def countdown(n):
...     while n > 0:
...         print('T-minus', n)
...         n -= 1
...     print('Blastoff!')
...
>>> import dis
>>> dis.dis(countdown)
  2           0 SETUP_LOOP              30 (to 32)
        >>    2 LOAD_FAST                0 (n)
              4 LOAD_CONST               1 (0)
              6 COMPARE_OP               4 (>)
              8 POP_JUMP_IF_FALSE       30

  3          10 LOAD_GLOBAL              0 (print)
             12 LOAD_CONST               2 ('T-minus')
             14 LOAD_FAST                0 (n)
             16 CALL_FUNCTION            2
             18 POP_TOP

  4          20 LOAD_FAST                0 (n)
             22 LOAD_CONST               3 (1)
             24 INPLACE_SUBTRACT
             26 STORE_FAST               0 (n)
             28 JUMP_ABSOLUTE            2
        >>   30 POP_BLOCK

  5     >>   32 LOAD_GLOBAL              0 (print)
             34 LOAD_CONST               4 ('Blastoff!')
             36 CALL_FUNCTION            1
             38 POP_TOP
             40 LOAD_CONST               0 (None)
             42 RETURN_VALUE
>>>
```

讨论¶

当你想要知道你的程序底层的运行机制的时候，`dis` 模块是很有用的。比如如果你想试着理解性能特征。 被 `dis()` 函数解析的原始字节码如下所示：

```
>>> countdown.__code__.co_code
b"x'\x00|\x00\x00d\x01\x00k\x04\x00r)\x00t\x00\x00d\x02\x00|\x00\x00\x83
\x02\x00\x01|\x00\x00d\x03\x008}\x00\x00q\x03\x00Wt\x00\x00d\x04\x00\x83
\x01\x00\x01d\x00\x00S"
>>>
```

如果你想自己解释这段代码，你需要使用一些在 `opcode` 模块中定义的常量。例如：

```
>>> c = countdown.__code__.co_code
>>> import opcode
>>> opcode.opname[c[0]]
'SETUP_LOOP'
>>> opcode.opname[c[2]]
'LOAD_FAST'
>>>
```

奇怪的是，在 `dis` 模块中并没有函数让你以编程方式很容易的来处理字节码。 不过，下面的生成器函数可以将原始字节码序列转换成 `opcodes` 和参数。

```
import opcode

def generate_opcodes(codebytes):
    extended_arg = 0
    i = 0
    n = len(codebytes)
    while i < n:
        op = codebytes[i]
        i += 1
        if op >= opcode.HAVE_ARGUMENT:
            oparg = codebytes[i] + codebytes[i+1]*256 + extended_arg
            extended_arg = 0
            i += 2
            if op == opcode.EXTENDED_ARG:
                extended_arg = oparg * 65536
                continue
        else:
            oparg = None
        yield (op, oparg)
```

使用方法如下：

```
>>> for op, oparg in generate_opcodes(countdown.__code__.co_code):
...     print(op, opcode.opname[op], oparg)
```

这种方式很少有人知道，你可以利用它替换任何你想要替换的函数的原始字节码。 下面我们用一个示例来演示整个过程：

```
>>> def add(x, y):
...     return x + y
...
>>> c = add.__code__
>>> c
<code object add at 0x1007beed0, file "<stdin>", line 1>
>>> c.co_code
b'|\x00\x00|\x01\x00\x17S'
>>>
>>> # Make a completely new code object with bogus byte code
>>> import types
>>> newbytecode = b'xxxxxxx'
>>> nc = types.CodeType(c.co_argcount, c.co_kwonlyargcount,
...     c.co_nlocals, c.co_stacksize, c.co_flags, newbytecode, c.co_consts,
...     c.co_names, c.co_varnames, c.co_filename, c.co_name,
...     c.co_firstlineno, c.co_lnotab)
>>> nc
<code object add at 0x10069fe40, file "<stdin>", line 1>
>>> add.__code__ = nc
>>> add(2,3)
Segmentation fault
```

你可以像这样耍大招让解释器奔溃。但是，对于编写更高级优化和元编程工具的程序员来讲， 他们可能真的需要重写字节码。本节最后的部分演示了这个是怎样做到的。你还可以参考另外一个类似的例子： this code on ActiveState

9.3 解除一个装饰器¶

问题¶

一个装饰器已经作用在一个函数上，你想撤销它，直接访问原始的未包装的那个函数。

解决方案¶

假设装饰器是通过 `@wraps` (参考9.2小节)来实现的，那么你可以通过访问 `__wrapped__` 属性来访问原始函数：

```
>>> @somedecorator
>>> def add(x, y):
...     return x + y
...
>>> orig_add = add.__wrapped__
>>> orig_add(3, 4)
7
>>>
```

讨论¶

直接访问未包装的原始函数在调试、内省和其他函数操作时是很有用的。 但是我们这里的方案仅仅适用于在包装器中正确使用了 `@wraps` 或者直接设置了 `__wrapped__` 属性的情况。

如果有多个包装器，那么访问 `__wrapped__` 属性的行为是不可预知的，应该避免这样做。 在Python3.3中，它会略过所有的包装层，比如，假如你有如下的代码：

```
from functools import wraps

def decorator1(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Decorator 1')
        return func(*args, **kwargs)
    return wrapper

def decorator2(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Decorator 2')
        return func(*args, **kwargs)
    return wrapper

@decorator1
@decorator2
def add(x, y):
    return x + y
```

下面我们在Python3.3下测试：

```
>>> add(2, 3)
Decorator 1
Decorator 2
5
>>> add.__wrapped__(2, 3)
5
>>>
```

下面我们在Python3.4下测试：

```
>>> add(2, 3)
Decorator 1
Decorator 2
5
```

```
>>> add.__wrapped__(2, 3)
Decorator 2
5
>>>
```

最后要说的是，并不是所有的装饰器都使用了 `@wraps`，因此这里的方案并不全部适用。特别的，内置的装饰器 `@staticmethod` 和 `@classmethod` 就没有遵循这个约定 (它们把原始函数存储在属性 `__func__` 中)。

9.4 定义一个带参数的装饰器¶

问题¶

你想定义一个可以接受参数的装饰器

解决方案¶

我们用一个例子详细阐述下接受参数的处理过程。 假设你想写一个装饰器，给函数添加日志功能，同时允许用户指定日志的级别和其他的选项。 下面是这个装饰器的定义和使用示例：

```
from functools import wraps
import logging

def logged(level, name=None, message=None):
    """
    Add logging to a function. level is the logging
    level, name is the logger name, and message is the
    log message. If name and message aren't specified,
    they default to the function's module and name.
    """
    def decorate(func):
        logname = name if name else func.__module__
        log = logging.getLogger(logname)
        logmsg = message if message else func.__name__

        @wraps(func)
        def wrapper(*args, **kwargs):
            log.log(level, logmsg)
            return func(*args, **kwargs)
        return wrapper
    return decorate

# Example use
@logged(logging.DEBUG)
def add(x, y):
    return x + y

@logged(logging.CRITICAL, 'example')
def spam():
    print('Spam!')
```

初看起来，这种实现看上去很复杂，但是核心思想很简单。 最外层的函数 `logged()` 接受参数并将它们作用在内部的装饰器函数上面。 内层的函数 `decorate()` 接受一个函数作为参数，然后在函数上面放置一个包装器。 这里的关键点是包装器是可以使用传递给 `logged()` 的参数的。

讨论¶

定义一个接受参数的包装器看上去比较复杂主要是因为底层的调用序列。特别的，如果你有下面这个代码：

```
@decorator(x, y, z)
def func(a, b):
    pass
```

装饰器处理过程跟下面的调用是等效的;

```
def func(a, b):
    pass
func = decorator(x, y, z)(func)
```

`decorator(x, y, z)` 的返回结果必须是一个可调用对象，它接受一个函数作为参数并包装它， 可以参考9.7小节中另外一个可接受参数的包装器例子。

9.5 可自定义属性的装饰器¶

问题¶

你想写一个装饰器来包装一个函数，并且允许用户提供参数在运行时控制装饰器行为。

解决方案¶

引入一个访问函数，使用 `nonlocal` 来修改内部变量。 然后这个访问函数被作为一个属性赋值给包装函数。

```
from functools import wraps, partial
import logging
# Utility decorator to attach a function as an attribute of obj
def attach_wrapper(obj, func=None):
    if func is None:
        return partial(attach_wrapper, obj)
    setattr(obj, func.__name__, func)
    return func

def logged(level, name=None, message=None):
    '''
    Add logging to a function. level is the logging
    level, name is the logger name, and message is the
    log message. If name and message aren't specified,
    they default to the function's module and name.
    '''
    def decorate(func):
        logname = name if name else func.__module__
        log = logging.getLogger(logname)
        logmsg = message if message else func.__name__

        @wraps(func)
        def wrapper(*args, **kwargs):
            log.log(level, logmsg)
            return func(*args, **kwargs)

        # Attach setter functions
        @attach_wrapper(wrapper)
        def set_level(newlevel):
            nonlocal level
            level = newlevel

        @attach_wrapper(wrapper)
        def set_message(newmsg):
            nonlocal logmsg
            logmsg = newmsg

        return wrapper

    return decorate

# Example use
@logged(logging.DEBUG)
def add(x, y):
    return x + y

@logged(logging.CRITICAL, 'example')
def spam():
    print('Spam!')
```

下面是交互环境下的使用例子：

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> add(2, 3)
DEBUG:__main__:add
```

```
5
>>> # Change the log message
>>> add.set_message('Add called')
>>> add(2, 3)
DEBUG:__main__:Add called
5
>>> # Change the log level
>>> add.set_level(logging.WARNING)
>>> add(2, 3)
WARNING:__main__:Add called
5
>>>
```

讨论¶

这一小节的关键点在于访问函数(如 `set_message()` 和 `set_level()`)，它们被作为属性赋给包装器。 每个访问函数允许使用 `nonlocal` 来修改函数内部的变量。

还有一个令人吃惊的地方是访问函数会在多层装饰器间传播(如果你的装饰器都使用了 `@functools.wraps` 注解)。 例如，假设你引入另外一个装饰器，比如9.2小节中的 `@timethis` ，像下面这样：

```
@timethis
@logged(logging.DEBUG)
def countdown(n):
    while n > 0:
        n -= 1
```

你会发现访问函数依旧有效：

```
>>> countdown(10000000)
DEBUG:__main__:countdown
countdown 0.8198461532592773
>>> countdown.set_level(logging.WARNING)
>>> countdown.set_message("Counting down to zero")
>>> countdown(10000000)
WARNING:__main__:Counting down to zero
countdown 0.8225970268249512
>>>
```

你还会发现即使装饰器像下面这样以相反的方向排放，效果也是一样的：

```
@logged(logging.DEBUG)
@timethis
def countdown(n):
    while n > 0:
        n -= 1
```

还能通过使用lambda表达式代码来让访问函数的返回不同的设定值：

```
@attach_wrapper(wrapper)
def get_level():
    return level

# Alternative
wrapper.get_level = lambda: level
```

一个比较难理解的地方就是对于访问函数的首次使用。例如，你可能会考虑另外一个方法直接访问函数的属性，如下：

```
@wraps(func)
def wrapper(*args, **kwargs):
    wrapper.log.log(wrapper.level, wrapper.logmsg)
    return func(*args, **kwargs)

# Attach adjustable attributes
wrapper.level = level
wrapper.logmsg = logmsg
```

```
wrapper.log = log
```

这个方法也可能正常工作，但前提是它必须是最外层的装饰器才行。 如果它的上面还有另外的装饰器(比如上面提到的 `@timethis` 例子)，那么它会隐藏底层属性，使得修改它们没有任何作用。 而通过使用访问函数就能避免这样的局限性。

最后提一点，这一小节的方案也可以作为9.9小节中装饰器类的另一种实现方法。

9.6 带可选参数的装饰器¶

问题¶

你想写一个装饰器，既可以不传参数给它，比如 `@decorator`， 也可以传递可选参数给它，比如 `@decorator(x,y,z)`。

解决方案¶

下面是9.5小节中日志装饰器的一个修改版本：

```
from functools import wraps, partial
import logging

def logged(func=None, *, level=logging.DEBUG, name=None, message=None):
    if func is None:
        return partial(logged, level=level, name=name, message=message)

    logname = name if name else func.__module__
    log = logging.getLogger(logname)
    logmsg = message if message else func.__name__

    @wraps(func)
    def wrapper(*args, **kwargs):
        log.log(level, logmsg)
        return func(*args, **kwargs)

    return wrapper

# Example use
@logged
def add(x, y):
    return x + y

@logged(level=logging.CRITICAL, name='example')
def spam():
    print('Spam!')
```

可以看到，`@logged` 装饰器可以同时不带参数或带参数。

讨论¶

这里提到的这个问题就是通常所说的编程一致性问题。 当我们使用装饰器的时候，大部分程序员习惯了要么不给它们传递任何参数，要么给它们传递确切参数。 其实从技术上来讲，我们可以定义一个所有参数都是可选的装饰器，就像下面这样：

```
@logged()
def add(x, y):
    return x+y
```

但是，这种写法并不符合我们的习惯，有时候程序员忘记加上后面的括号会导致错误。 这里我们向你展示了如何以一致的编程风格来同时满足没有括号和有括号两种情况。

为了理解代码是如何工作的，你需要非常熟悉装饰器是如何作用到函数上以及它们的调用规则。 对于一个像下面这样的简单装饰器：

```
# Example use
@logged
def add(x, y):
    return x + y
```

这个调用序列跟下面等价：

```
def add(x, y):
    return x + y

add = logged(add)
```

这时候，被装饰函数会被当做第一个参数直接传递给 `logged` 装饰器。 因此，`logged()` 中的第一个参数就是被包装函数本身。所有其他参数都必须有默认值。

而对于一个下面这样有参数的装饰器：

```
@logged(level=logging.CRITICAL, name='example')
def spam():
    print('Spam!')
```

调用序列跟下面等价：

```
def spam():
    print('Spam!')
spam = logged(level=logging.CRITICAL, name='example')(spam)
```

初始调用 `logged()` 函数时，被包装函数并没有传递进来。 因此在装饰器内，它必须是可选的。这个反过来会迫使其他参数必须使用关键字来指定。 并且，但这些参数被传递进来后，装饰器要返回一个接受一个函数参数并包装它的函数(参考9.5小节)。 为了这样做，我们使用了一个技巧，就是利用 `functools.partial` 。 它会返回一个未完全初始化的自身，除了被包装函数外其他参数都已经确定下来了。 可以参考7.8小节获取更多 `partial()` 方法的知识。

9.7 利用装饰器强制函数上的类型检查¶

问题¶

作为某种编程规约，你想在对函数参数进行强制类型检查。

解决方案¶

在演示实际代码前，先说明我们的目标：能对函数参数类型进行断言，类似下面这样：

```
>>> @typeassert(int, int)
... def add(x, y):
...     return x + y
...
>>>
>>> add(2, 3)
5
>>> add(2, 'hello')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "contract.py", line 33, in wrapper
TypeError: Argument y must be <class 'int'>
>>>
```

下面是使用装饰器技术来实现 `@typeassert`：

```
from inspect import signature
from functools import wraps

def typeassert(*ty_args, **ty_kwargs):
    def decorate(func):
        # If in optimized mode, disable type checking
        if not __debug__:
            return func

        # Map function argument names to supplied types
        sig = signature(func)
        bound_types = sig.bind_partial(*ty_args, **ty_kwargs).arguments

        @wraps(func)
        def wrapper(*args, **kwargs):
            bound_values = sig.bind(*args, **kwargs)
            # Enforce type assertions across supplied arguments
            for name, value in bound_values.arguments.items():
                if name in bound_types:
                    if not isinstance(value, bound_types[name]):
                        raise TypeError(
                            'Argument {} must be {}'.format(name, bound_types[name])
                            )
            return func(*args, **kwargs)
        return wrapper
    return decorate
```

可以看出这个装饰器非常灵活，既可以指定所有参数类型，也可以只指定部分。 并且可以通过位置或关键字来指定参数类型。下面是使用示例：

```
>>> @typeassert(int, z=int)
... def spam(x, y, z=42):
...     print(x, y, z)
...
>>> spam(1, 2, 3)
1 2 3
>>> spam(1, 'hello', 3)
1 hello 3
>>> spam(1, 'hello', 'world')
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
File "contract.py", line 33, in wrapper
TypeError: Argument z must be <class 'int'>
>>>
```

讨论¶

这节是高级装饰器示例，引入了很多重要的概念。

首先，装饰器只会在函数定义时被调用一次。 有时候你去掉装饰器的功能，那么你只需要简单的返回被装饰函数即可。 下面的代码中，如果全局变量 `__debug__` 被设置成了False(当你使用-O或-OO参数的优化模式执行程序时)， 那么就直接返回未修改过的函数本身：

```
def decorate(func):
    # If in optimized mode, disable type checking
    if not __debug__:
        return func
```

其次，这里还对被包装函数的参数签名进行了检查，我们使用了 `inspect.signature()` 函数。 简单来讲，它运行你提取一个可调用对象的参数签名信息。例如：

```
>>> from inspect import signature
>>> def spam(x, y, z=42):
...     pass
...
>>> sig = signature(spam)
>>> print(sig)
(x, y, z=42)
>>> sig.parameters
mappingproxy(OrderedDict([('x', <Parameter at 0x10077a050 'x'>),
('y', <Parameter at 0x10077a158 'y'>), ('z', <Parameter at 0x10077a1b0 'z'>)]))
>>> sig.parameters['z'].name
'z'
>>> sig.parameters['z'].default
42
>>> sig.parameters['z'].kind
<_ParameterKind: 'POSITIONAL_OR_KEYWORD'>
>>>
```

装饰器的开始部分，我们使用了 `bind_partial()` 方法来执行从指定类型到名称的部分绑定。 下面是例子演示：

```
>>> bound_types = sig.bind_partial(int,z=int)
>>> bound_types
<inspect.BoundArguments object at 0x10069bb50>
>>> bound_types.arguments
OrderedDict([('x', <class 'int'>), ('z', <class 'int'>)])
>>>
```

在这个部分绑定中，你可以注意到缺失的参数被忽略了(比如并没有对y进行绑定)。 不过最重要的是创建了一个有序字典 `bound_types.arguments` 。 这个字典会将参数名以函数签名中相同顺序映射到指定的类型值上面去。 在我们的装饰器例子中，这个映射包含了我们要强制指定的类型断言。

在装饰器创建的实际包装函数中使用到了 `sig.bind()` 方法。 `bind()` 跟 `bind_partial()` 类似，但是它不允许忽略任何参数。因此有了下面的结果：

```
>>> bound_values = sig.bind(1, 2, 3)
>>> bound_values.arguments
OrderedDict([('x', 1), ('y', 2), ('z', 3)])
>>>
```

使用这个映射我们可以很轻松的实现我们的强制类型检查：

```
>>> for name, value in bound_values.arguments.items():
...     if name in bound_types.arguments:
...         if not isinstance(value, bound_types.arguments[name]):
...             raise TypeError()
```

```
...
>>>
```

不过这个方案还有点小瑕疵，它对于有默认值的参数并不适用。 比如下面的代码可以正常工作，尽管items的类型是错误的：

```
>>> @typeassert(int, list)
... def bar(x, items=None):
...     if items is None:
...         items = []
...     items.append(x)
...     return items
>>> bar(2)
[2]
>>> bar(2,3)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "contract.py", line 33, in wrapper
TypeError: Argument items must be <class 'list'>
>>> bar(4, [1, 2, 3])
[1, 2, 3, 4]
>>>
```

最后一点是关于适用装饰器参数和函数注解之间的争论。 例如，为什么不像下面这样写一个装饰器来查找函数中的注解呢？

```
@typeassert
def spam(x:int, y, z:int = 42):
    print(x,y,z)
```

一个可能的原因是如果使用了函数参数注解，那么就被限制了。 如果注解被用来做类型检查就不能做其他事情了。而且 `@typeassert` 不能再用于使用注解做其他事情的函数了。 而使用上面的装饰器参数灵活性大多了，也更加通用。

可以在PEP 362以及 `inspect` 模块中找到更多关于函数参数对象的信息。在9.16小节还有另外一个例子。

9.8 将装饰器定义为类的一部分¶

问题¶

你想在类中定义装饰器，并将其作用在其他函数或方法上。

解决方案¶

在类里面定义装饰器很简单，但是你首先要确认它的使用方式。比如到底是作为一个实例方法还是类方法。 下面我们用例子来阐述它们的不同：

```
from functools import wraps

class A:
    # Decorator as an instance method
    def decorator1(self, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print('Decorator 1')
            return func(*args, **kwargs)
        return wrapper

    # Decorator as a class method
    @classmethod
    def decorator2(cls, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print('Decorator 2')
            return func(*args, **kwargs)
        return wrapper
```

下面是一使用例子：

```
# As an instance method
a = A()
@a.decorator1
def spam():
    pass
# As a class method
@A.decorator2
def grok():
    pass
```

仔细观察可以发现一个是实例调用，一个是类调用。

讨论¶

在类中定义装饰器初看上去好像很奇怪，但是在标准库中有很多这样的例子。 特别的，`@property` 装饰器实际上是一个类，它里面定义了三个方法 `getter()`, `setter()`, `deleter()` ,每一个方法都是一个装饰器。例如：

```
class Person:
    # Create a property instance
    first_name = property()

    # Apply decorator methods
    @first_name.getter
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value
```

它为什么要这么定义的主要原因是各种不同的装饰器方法会在关联的 `property` 实例上操作它的状态。 因此，任何时候只要你碰到需要在装饰器中记录或绑定信息，那么这不失为一种可行方法。

在类中定义装饰器有个难理解的地方就是对于额外参数 `self` 或 `cls` 的正确使用。 尽管最外层的装饰器函数比如 `decorator1()` 或 `decorator2()` 需要提供一个 `self` 或 `cls` 参数， 但是在两个装饰器内部被创建的 `wrapper()` 函数并不需要包含这个 `self` 参数。 你唯一需要这个参数是在你确实要访问包装器中这个实例的某些部分的时候。其他情况下都不用去管它。

对于类里面定义的包装器还有一点比较难理解，就是在涉及到继承的时候。 例如，假设你想让在A中定义的装饰器作用在子类B中。你需要像下面这样写：

```
class B(A):
    @A.decorator2
    def bar(self):
        pass
```

也就是说，装饰器要被定义成类方法并且你必须显式的使用父类名去调用它。 你不能使用 `@B.decorator2` ，因为在方法定义时，这个类B还没有被创建。

9.9 将装饰器定义为类¶

问题¶

你想使用一个装饰器去包装函数，但是希望返回一个可调用的实例。 你需要让你的装饰器可以同时工作在类定义的内部和外部。

解决方案¶

为了将装饰器定义成一个实例，你需要确保它实现了 `__call__()` 和 `__get__()` 方法。 例如，下面的代码定义了一个类，它在其他函数上放置一个简单的记录层：

```
import types
from functools import wraps

class Profiled:
    def __init__(self, func):
        wraps(func)(self)
        self.ncalls = 0

    def __call__(self, *args, **kwargs):
        self.ncalls += 1
        return self.__wrapped__(*args, **kwargs)

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return types.MethodType(self, instance)
```

你可以将它当做一个普通的装饰器来使用，在类里面或外面都可以：

```
@Profiled
def add(x, y):
    return x + y

class Spam:
    @Profiled
    def bar(self, x):
        print(self, x)
```

在交互环境中的使用示例：

```
>>> add(2, 3)
5
>>> add(4, 5)
9
>>> add.ncalls
2
>>> s = Spam()
>>> s.bar(1)
<__main__.Spam object at 0x10069e9d0> 1
>>> s.bar(2)
<__main__.Spam object at 0x10069e9d0> 2
>>> s.bar(3)
<__main__.Spam object at 0x10069e9d0> 3
>>> Spam.bar.ncalls
3
```

讨论¶

将装饰器定义成类通常是很简单的。但是这里还是有一些细节需要解释下，特别是当你想将它作用在实例方法上的时候。

首先，使用 `functools.wraps()` 函数的作用跟之前还是一样，将被包装函数的元信息复制到可调用实例中去。

其次，通常很容易会忽视上面的 `__get__()` 方法。如果你忽略它，保持其他代码不变再次运行， 你会发现当你去调用被装饰实例方法时出现很奇怪的问题。例如：

```
>>> s = Spam()
>>> s.bar(3)
Traceback (most recent call last):
...
TypeError: bar() missing 1 required positional argument: 'x'
```

出错原因是当方法函数在一个类中被查找时，它们的 `__get__()` 方法依据描述器协议被调用， 在8.9小节已经讲述过描述器协议了。在这里， `__get__()` 的目的是创建一个绑定方法对象 (最终会给这个方法传递self参数)。下面是一个例子来演示底层原理：

```
>>> s = Spam()
>>> def grok(self, x):
...     pass
...
>>> grok.__get__(s, Spam)
<bound method Spam.grok of <__main__.Spam object at 0x100671e90>>
>>>
```

`__get__()` 方法是为了确保绑定方法对象能被正确的创建。 `type.MethodType()` 手动创建一个绑定方法来使用。只有当实例被使用的时候绑定方法才会被创建。 如果这个方法是在类上面来访问， 那么 `__get__()` 中的instance参数会被设置成None并直接返回 `Profiled` 实例本身。 这样的话我们就可以提取它的 `ncalls` 属性了。

如果你想避免一些混乱，也可以考虑另外一个使用闭包和 `nonlocal` 变量实现的装饰器，这个在9.5小节有讲到。例如：

```
import types
from functools import wraps

def profiled(func):
    ncalls = 0
    @wraps(func)
    def wrapper(*args, **kwargs):
        nonlocal ncalls
        ncalls += 1
        return func(*args, **kwargs)
    wrapper.ncalls = lambda: ncalls
    return wrapper

# Example
@profiled
def add(x, y):
    return x + y
```

这个方式跟之前的效果几乎一样，除了对于 `ncalls` 的访问现在是通过一个被绑定为属性的函数来实现，例如：

```
>>> add(2, 3)
5
>>> add(4, 5)
9
>>> add.ncalls()
2
>>>
```

第九章：元编程¶

软件开发领域中最经典的口头禅就是“don’t repeat yourself”。 也就是说，任何时候当你的程序中存在高度重复(或者是通过剪切复制)的代码时，都应该想想是否有更好的解决方案。 在Python当中，通常都可以通过元编程来解决这类问题。 简而言之，元编程就是关于创建操作源代码(比如修改、生成或包装原来的代码)的函数和类。 主要技术是使用装饰器、类装饰器和元类。不过还有一些其他技术， 包括签名对象、使用 `exec()` 执行代码以及对内部函数和类的反射技术等。 本章的主要目的是向大家介绍这些元编程技术，并且给出实例来演示它们是怎样定制化你的源代码行为的。

Contents:

- 9.1 在函数上添加包装器
- 9.2 创建装饰器时保留函数元信息
- 9.3 解除一个装饰器
- 9.4 定义一个带参数的装饰器
- 9.5 可自定义属性的装饰器
- 9.6 带可选参数的装饰器
- 9.7 利用装饰器强制函数上的类型检查
- 9.8 将装饰器定义为类的一部分
- 9.9 将装饰器定义为类
- 9.10 为类和静态方法提供装饰器
- 9.11 装饰器为被包装函数增加参数
- 9.12 使用装饰器扩充类的功能
- 9.13 使用元类控制实例的创建
- 9.14 捕获类的属性定义顺序
- 9.15 定义有可选参数的元类
- 9.16 *args和**kwargs的强制参数签名
- 9.17 在类上强制使用编程规约
- 9.18 以编程方式定义类
- 9.19 在定义的时候初始化类的成员
- 9.20 利用函数注解实现方法重载
- 9.21 避免重复的属性方法
- 9.22 定义上下文管理器的简单方法
- 9.23 在局部变量域中执行代码
- 9.24 解析与分析Python源码
- 9.25 拆解Python字节码