听说你会 Python ？

2016-11-18 编程 PYTHON #PYTHON, #协程, #编程, #编程技巧

前言

最近觉得 Python 太“简单了”，于是在师父川爷面前放肆了一把：“我觉得 Python 是世界上最简单的语言！”。于是川爷嘴角闪过了一丝轻蔑的微笑（内心 OS：Naive！，作为一个 Python 开发者，我必须要给你一点人生经验，不然你不知道天高地厚！）于是川爷给我了一份满分 100 分的题，然后这篇文章就是记录下做这套题所踩过的坑。

1.列表生成器

描述

下面的代码会报错，为什么？

```
class A(object):
    x = 1
    gen = (x for _ in xrange(10))  # gen=(x for _ in range(10))


if __name__ == "__main__":
    print(list(A.gen))
```

答案

这个问题是变量作用域问题，在 `gen=(x for _ in xrange(10))` 中 `gen` 是一个 `generator` ,在 `generator` 中变量有自己的一套作用域，与其余作用域空间相互隔离。因此，将会出现这样的 `NameError: name 'x' is not defined` 的问题，那么解决方案是什么呢？答案是：用 lambda 。

```
class A(object):
    x = 1
    gen = (lambda x: (x for _ in xrange(10)))(x)  # gen=(x for _ in range(10))


if __name__ == "__main__":
    print(list(A.gen))
```

2.装饰器

描述

我想写一个类装饰器用来度量函数/方法运行时间

```
import time

class Timeit(object):
    def __init__(self, func):
        self._wrapped = func

    def __call__(self, *args, **kws):
        start_time = time.time()
```

```
        result = self._wrapped(*args, **kws)
        print("elapsed time is %s " % (time.time() - start_time))
        return result
```

这个装饰器能够运行在普通函数上：

```
@Timeit
def func():
    time.sleep(1)
    return "invoking function func"


if __name__ == '__main__':
    func()  # output: elapsed time is 1.00044410133
```

但是运行在方法上会报错，为什么？

```
class A(object):
    @Timeit
    def func(self):
        time.sleep(1)
        return 'invoking method func'


if __name__ == '__main__':
    a = A()
    a.func()  # Boom!
```

如果我坚持使用类装饰器，应该如何修改？

答案

使用类装饰器后，在调用 `func` 函数的过程中其对应的 instance 并不会传递给 `__call__` 方法，造成其 `mehtod unbound` ,那么解决方法是什么呢？描述符赛高

```
class Timeit(object):
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print('invoking Timer')

    def __get__(self, instance, owner):
        return lambda *args, **kwargs: self.func(instance, *args, **kwargs)
```

3.Python 调用机制

描述

我们知道 `__call__` 方法可以用来重载圆括号调用，好的，以为问题就这么简单？Naive！

```
class A(object):
    def __call__(self):
        print("invoking __call__ from A!")


if __name__ == "__main__":
```

```
7       a = A()
8       a()  # output: invoking __call__ from A
```

现在我们可以看到 `a()` 似乎等价于 `a.__call__()` ,看起来很 Easy 对吧，好的，我现在想作死，又写出了如下的代码，

```
1  a.__call__ = lambda: "invoking __call__ from lambda"
2  a.__call__()
3  # output:invoking __call__ from lambda
4  a()
5
6
7  # output:invoking __call__ from A!
```

请大佬们解释下，为什么 `a()` 没有调用出 `a.__call__()` (此题由 USTC 王子博前辈提出)

答案

原因在于，在 Python 中，新式类（ new class)的内建特殊方法，和实例的属性字典是相互隔离的，具体可以看看 Python 官方文档对于这一情况的说明

> For new-style classes, implicit invocations of special methods are only guaranteed to work correctly if defined on an object's type, not in the object's instance dictionary. That behaviour is the reason why the following code raises an exception (unlike the equivalent example with old-style classes):

同时官方也给出了一个例子：

```
1  class C(object):
2      pass
3
4
5  c = C()
6  c.__len__ = lambda: 5
7  len(c)
8
9
10 # Traceback (most recent call last):
11 #  File "<stdin>", line 1, in <module>
12 # TypeError: object of type 'C' has no len()
```

回到我们的例子上来，当我们在执行 `a.__call__=lambda:"invoking __call__ from lambda"` 时，的确在我们在 `a.__dict__` 中新增加了一个 key 为 `__call__` 的 item，但是当我们执行 `a()` 时，因为涉及特殊方法的调用，因此我们的调用过程不会从 `a.__dict__` 中寻找属性，而是从 `tyee(a).__dict__` 中寻找属性。因此，就会出现如上所述的情况。

4.描述符

描述

我想写一个 Exam 类，其属性 math 为 [0,100] 的整数，若赋值时不在此范围内则抛出异常，我决定用描述符来实现这个需求。

```
1  class Grade(object):
2      def __init__(self):
3          self._score = 0
4
5      def __get__(self, instance, owner):
6          return self._score
7
8      def __set__(self, instance, value):
```

```
        if 0 <= value <= 100:
            self._score = value
        else:
            raise ValueError('grade must be between 0 and 100')


class Exam(object):
    math = Grade()

    def __init__(self, math):
        self.math = math


if __name__ == '__main__':
    niche = Exam(math=90)
    print(niche.math)
    # output : 90
    snake = Exam(math=75)
    print(snake.math)
    # output : 75
    snake.math = 120
    # output: ValueError:grade must be between 0 and 100!
```

看起来一切正常。不过这里面有个巨大的问题，尝试说明是什么问题
为了解决这个问题，我改写了 Grade 描述符如下：

```
class Grad(object):
    def __init__(self):
        self._grade_pool = {}

    def __get__(self, instance, owner):
        return self._grade_pool.get(instance, None)

    def __set__(self, instance, value):
        if 0 <= value <= 100:
            _grade_pool = self.__dict__.setdefault('_grade_pool', {})
            _grade_pool[instance] = value
        else:
            raise ValueError("fuck")
```

不过这样会导致更大的问题，请问该怎么解决这个问题？

答案

1.第一个问题的其实很简单，如果你再运行一次 `print(niche.math)` 你就会发现，输出值是 `120` ，那么这是为什么呢？这就要先从 Python 的调用机制说起了。我们如果调用一个属性，那么其顺序是优先从实例的 `__dict__` 里查找，然后如果没有查找到的话，那么一次查询类字典，父类字典，直到彻底查不到为止。好的，现在回到我们的问题，我们发现，在我们的类 `Exam` 中，其 `self.math` 的调用过程是，首先在实例化后的实例的 `__dict__` 中进行查找，没有找到，接着往上一级，在我们的类 `Exam` 中进行查找，好的找到了，返回。那么这意味着，我们对于 `self.math` 的所有操作都是对于类变量 `math` 的操作。因此造成变量污染的问题。那么该则怎么解决呢？很多同志可能会说，恩，在 `__set__` 函数中将值设置到具体的实例字典不就行了。
那么这样可不可以呢？答案是，很明显不得行啊，至于为什么，就涉及到我们 Python 描述符的机制了，描述符指的是实现了描述符协议的特殊的类，三个描述符协议指的是 `__get__` ，‘**set**‘， `__delete__` 以及 Python 3.6 中新增的 `__set_name__` 方法，其中实现了 `__get__` 以及 `__set__` / `__delete__` / `__set_name__` 的是 **Data descriptors** ，而只实现了 `__get__` 的是 `Non-Data descriptor` 。那么有什么区别呢，前面说了， **我们如果调用一个属性，那么其顺序是优先从实例的 `__dict__` 里查找，然后如果没有查找到的话，那么一次查询类字典，父类字典，直到彻底查不到为止。** 但是，这里没有考虑描述符的因素进去，如果将描述符因素考虑进去，那么正确的表述应该是**我们如果调用一个属性，那么其顺序是优先从实例的 `__dict__` 里查找，然后如果没有查找到的话，那么一次查询类字典，父类字典，直到彻底查不到为止。其中如果在类实例字典中的该属性是一个 `Data descriptors` ，那么无论实例字典中存在该属性与否，无条件走描述符协议进行调用，在类实例字典中的该属性是一个 `Non-Data descriptors` ，那么优先调用实例字典中的属性值而不触发描述符协议，如果实例字典中不存在该属性值，那么触发 `Non-Data descriptor` 的描述符协议。**回到之前的问题，我们即使在 `__set__` 将具体的属性写入实例字典中，但是由于类字典中存在着 `Data descriptors` ，因此，我们在调用 `math` 属性时，依旧会触发描述符协议。

2.经过改良的做法，利用 `dict` 的 key 唯一性，将具体的值与实例进行绑定，但是同时带来了内存泄露的问题。那么为什么会造成内存泄露呢，首先复习下我们的 `dict` 的特性， `dict` 最重要的一个特性，就是凡可 hash 的对象皆可为 key ， `dict` 通过利用的 hash 值的唯一性（严格意义上来讲并不是唯一，而是其 hash 值碰撞几率极小，近似认定其唯一）来保证 key 的不重复性，同时（敲黑板，重点来了）， `dict` 中的 `key` 引用是强引用类型，会造成对应

对象的引用计数的增加，可能造成对象无法被 gc ，从而产生内存泄露。那么这里该怎么解决呢？两种方法

第一种：

```
class Grad(object):
    def __init__(self):
        import weakref
        self._grade_pool = weakref.WeakKeyDictionary()

    def __get__(self, instance, owner):
        return self._grade_pool.get(instance, None)

    def __set__(self, instance, value):
        if 0 <= value <= 100:
            _grade_pool = self.__dict__.setdefault('_grade_pool', {})
            _grade_pool[instance] = value
        else:
            raise ValueError("fuck")
```

weakref 库中的 `WeakKeyDictionary` 所产生的字典的 key 对于对象的引用是弱引用类型，其不会造成内存引用计数的增加，因此不会造成内存泄露。同理，如果我们为了避免 value 对于对象的强引用，我们可以使用 `WeakValueDictionary` 。

第二种：在 Python 3.6 中，实现的 PEP 487 提案，为描述符新增加了一个协议，我们可以用其来绑定对应的对象：

```
class Grad(object):
    def __get__(self, instance, owner):
        return instance.__dict__[self.key]

    def __set__(self, instance, value):
        if 0 <= value <= 100:
            instance.__dict__[self.key] = value
        else:
            raise ValueError("fuck")

    def __set_name__(self, owner, name):
        self.key = name
```

这道题涉及的东西比较多，这里给出一点参考链接，invoking-descriptors , Descriptor HowTo Guide , PEP 487 , what`s new in Python 3.6 。

5.Python 继承机制

描述

试求出以下代码的输出结果。

```
class Init(object):
    def __init__(self, value):
        self.val = value


class Add2(Init):
    def __init__(self, val):
        super(Add2, self).__init__(val)
        self.val += 2


class Mul5(Init):
    def __init__(self, val):
        super(Mul5, self).__init__(val)
        self.val *= 5


class Pro(Mul5, Add2):
```

```
    pass


class Incr(Pro):
    csup = super(Pro)

    def __init__(self, val):
        self.csup.__init__(val)
        self.val += 1


p = Incr(5)
print(p.val)
```

答案

输出是 36 ，具体可以参考 New-style Classes , multiple-inheritance

6. Python 特殊方法

描述

我写了一个通过重载 **new** 方法来实现单例模式的类。

```
class Singleton(object):
    _instance = None

    def __new__(cls, *args, **kwargs):
        if cls._instance:
            return cls._instance
        cls._isntance = cv = object.__new__(cls, *args, **kwargs)
        return cv


sin1 = Singleton()
sin2 = Singleton()
print(sin1 is sin2)
# output: True
```

现在我有一堆类要实现为单例模式，所以我打算照葫芦画瓢写一个元类，这样可以让代码复用：

```
class SingleMeta(type):
    def __init__(cls, name, bases, dict):
        cls._instance = None
        __new__o = cls.__new__

        def __new__(cls, *args, **kwargs):
            if cls._instance:
                return cls._instance
            cls._instance = cv = __new__o(cls, *args, **kwargs)
            return cv

        cls.__new__ = __new__o


class A(object):
    __metaclass__ = SingleMeta


a1 = A()  # what`s the fuck
```

哎呀，好气啊，为啥这会报错啊，我明明之前用这种方法给 `__getattribute__` 打补丁的，下面这段代码能够捕获一切属性调用并打印参数

```
class TraceAttribute(type):
    def __init__(cls, name, bases, dict):
        __getattribute__o = cls.__getattribute__

        def __getattribute__(self, *args, **kwargs):
            print('__getattribute__:', args, kwargs)
            return __getattribute__o(self, *args, **kwargs)

        cls.__getattribute__ = __getattribute__


class A(object):  # Python 3 是 class A(object,metaclass=TraceAttribute):
    __metaclass__ = TraceAttribute
    a = 1
    b = 2


a = A()
a.a
# output: __getattribute__:('a',){}
a.b
```

试解释为什么给 **getattribute** 打补丁成功，而 **new** 打补丁失败。
如果我坚持使用元类给 **new** 打补丁来实现单例模式，应该怎么修改？

答案

其实这是最气人的一点，类里的 `__new__` 是一个 `staticmethod` 因此替换的时候必须以 `staticmethod` 进行替换。答案如下：

```
class SingleMeta(type):
    def __init__(cls, name, bases, dict):
        cls._instance = None
        __new__o = cls.__new__

        @staticmethod
        def __new__(cls, *args, **kwargs):
            if cls._instance:
                return cls._instance
            cls._instance = cv = __new__o(cls, *args, **kwargs)
            return cv

        cls.__new__ = __new__o


class A(object):
    __metaclass__ = SingleMeta


print(A() is A())  # output: True
```

结语

感谢师父大人的一套题让我开启新世界的大门，恩，博客上没法艾特，只能传递心意了。说实话 Python 的动态特性可以让其用众多 `black magic` 去实现一些很舒服的功能，当然这也对我们对语言特性及坑的掌握也变得更严格了，愿各位 Pythoner 没事阅读官方文档，早日达到**装逼如风，常伴吾身**的境界。

OLDER
Swift 声明式程序设计

RECENT

编程 PYTHON
听说你会 PYTHON ？
2016-11-18

编程 翻译
SWIFT 声明式程序设计
2016-10-26

编程 PYTHON
PYTHON 描述符入门指北
2016-10-12

编程 翻译
SWIFT 3 中的函数参数命名规范指北
2016-10-09

编程 PYTHON
聊聊 PYTHON 中生成器和协程那点事儿
2016-09-11

CATEGORIES

- 编程 (14)
  - Python (5)
  - 翻译 (9)

ARCHIVES

- November 2016 (1)
- October 2016 (3)
- September 2016 (2)
- August 2016 (3)
- July 2016 (5)

TAGS

- Flask (2)
- Objective-C (1)
- Python (5)
- Swift (8)
- iOS (8)
- 前端 (1)
- 协程 (3)
- 掘金翻译计划 (1)
- 源码阅读 (1)
- 编程 (14)
- 编程技巧 (3)

TAG CLOUD

Flask Objective-C Python Swift iOS 前端 协程 掘金翻译计划 源码阅读 编程 编程技巧

LINKS

- Hexo
- jianghao
- 小旋风
- allen
- 小天
- 猴哥
- 我心中尚未崩坏的地方
- Jing's Blog

- 鳗鱼鱼
- gayhub star过万的奇迹哥是我男票之一