Instancias únicas con *Python*

Cristian Ramírez
rvcristiand@unal.edu.co

24 de octubre de 2019

Resumen

Se presenta una *metaclase* que no le permite a las *clases creadas* con dicha metaclase *a*) crear *objetos* con los mismos *argumentos* con los que otros objetos de la misma clase han sido *instanciados* ni *b*) que los *atributos* de los objetos ya instanciados tomen los mismos valores de otros objetos de la misma clase.

Palabras clave— instancias únicas, memorización, metaclase, clase, objeto, atributo.

Índice

Algoritmos

1. Introducción

En ciertos *programas de computador* es necesario *instanciar objetos* cuyos valores de sus *atributos* sean diferentes a los de otros objetos de la misma *clase*. Por ejemplo, supóngase que se quiere almacenar coordenadas espaciales únicas, es decir, que dos objetos no describan el mismo punto espacial.

A continuación se presenta una serie de implementaciones que intentar solucionar el problema, donde se describen sus características, para así llegar a la *metaclase UniqueInstances*.

1.1. Primera implementación: *programación orientada a objetos.* Primer round

Esto se puede llevar a cabo *creando* una clase la cual instancie objetos con tres atributos. En el algoritmo 1.1 se presenta dicha implementación usando *Python*.

```python
class Coordinate:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
```

Algoritmo 1.1: Clase *Coordinate* que instancia objetos con tres atributos (x, y y z).

Sin embargo, la clase *Coordinate* permite instanciar varios objetos con los mismos atributos

```python
coord1 = Coordinate(0, 0, 0)
coord2 = Coordinate(0, 0, 0)  # obj mismos atributos otros objs
```

1.2. Segunda implementación: *programación procedimental*

Para evitar crear objetos con los mismos atributos, se debe llevar el registro de los *argumentos* con los que se van creando los objetos. En el algoritmo 1.2 se implementa la *función createUniqueCoordinates*, la cual permite instanciar objetos siempre y cuando los argumentos de entrada no se encuentren en el *diccionario coordianteRecord*.

```
coordinateRecord = {}

def createUniqueCoordinates(x, y, z):
    instanceAttr = (x, y, z)
    if instanceAttr not in coordinateRecord:
        coordinateRecord[instanceAttr] = Coordinate(x, y, z)
        return coordinateRecord[instanceAttr]
    else:
        print("La coordenada ya existe. No se creó el objeto")
        return None

coord1 = createUniqueCoordinates(0, 0, 0)
coord2 = createUniqueCoordinates(0, 0, 0)  # >>> La coordenada
                                           #     ya existe...
```

Algoritmo 1.2: Función *createUniqueCoordinates* que no permite crear objetos *tipo Coordinate* con los mismos argumentos.

Aunque la función *createUniqueCoordinates* no permite la instanciación de objetos de la clase *Coordinate* con los mismos argumentos con los que otros objetos fueron creados, aun se permite que un objeto *cambie* de manera tal que tenga los mismos atributos de otros objetos ya creados.

```
coord2 = createUniqueCoordinates(1, 1, 1)
coord2.x = coord2.y = coord2.z = 0  # >>> obj mismos atributos
                                    #     otros objs
```

1.3. Tercera implementación: *programación orientada a objetos.* Segundo round

Para evitar que un objeto cambie de manera tal que tenga los mismos atributos de otros objetos ya creados, se debe *verificar* el registro de los argumentos con los que se van creando los objetos antes de que estos cambien. Esto implica *actualizar* dicho registro.

En el algoritmo 1.3 se implementa la clase *UniqueInstances*, la cual debe ser heredada por otras clases, para que no les permita instanciar objetos cuando los argumentos de entrada se encuentren en el *set instancesAttrs*. Así mismo, los objetos instanciados con la *clase hija* no se pueden modificar si los nuevos atributos se encuentran en dicho *set*.

```
class UniqueInstances:
    __slots__ = []
    instancesAttrs = set()

    def __new__(cls, *args):
        assert len(args) == len(cls.__slots__)
        if  args not in cls.instancesAttrs:
            cls.instancesAttrs.add(args)

            return super(UniqueInstances, cls).__new__(cls)
        else:
            print("Warning: " +
                  "There is another instance of the class " +
                  "'{}'".format(cls.__name__) +
                  " with the same attibutes. The object was not
                   ↪  created.")

            return None

    def __init__(self, *args):
        for key, value in zip(self.__slots__, args):
            setattr(self, key, value)

    def __setattr__(self, key, value):
        if hasattr(self, key):
            instanceAttrs = tuple((getattr(self, _key) if _key !=
             ↪  key else value
                                   for _key in self.__slots__))
            if instanceAttrs not in
             ↪  self.__class__.instancesAttrs:
                self.__class__.instancesAttrs.\
                    remove(self.instanceAttrs)
                self.__class__.instancesAttrs.add(instanceAttrs)
            else:
                print("Warning: " +
                      "There is another instance of the class " +
                      "'{}'".format(self.__class__.__name__) +
                      " with the same attibutes. The object was
                       ↪  not changed.")

                return None

        super(UniqueInstances, self).__setattr__(key, value)

    @property
    def instanceAttrs(self):
```

```
43          return tuple(getattr(self, key) for key in
            ↪  self.__slots__)
44
45      def __repr__(self):
46          return "{}{}".format(self.__class__.__name__,
            ↪  str(self.instanceAttrs))
47
48      def __del__(self):
49          self.__class__.instancesAttrs.remove(self.instanceAttrs)
```

Algoritmo 1.3: Clase *UniqueInstances* que no le permite a las clases que heredan de esta clase crear objetos con los mismos argumentos ni que dichos objetos cambien para tomar los valores de otro objeto de la misma clase.

Dicha clase hace uso de varias herramientas de *programación orientada a objetos* las cuales se detallan a continuación.

1.3.1. __slots__

En Python, todos los objetos tienen por defecto un atributo llamado $\_\_dict\_\_$, el cual permite que dichos objetos puedan tener un número indeterminado de atributos, es decir

```
class MyClass:
    pass

myObj = MyClass()
myObj.x = 1; myObj.y = 2; myObj.z = 3

print(myObj.__dict__)  # {'x': 1, 'y': 2, 'z': 3}
```

Aunque muy flexible, dicha habilidad consume bastante memoria, más aún si se van a inicializar muchas instancias de esta clase. Por muchas se debe entender crear miles o millones de objetos.

Si se sabe de antemano la cantidad de atributos que tiene una instancia, éstas se pueden definir en la variable $\_\_slots\_\_$, evitando que dichas instancias tengan la variable $\_\_dict\_\_$, lo que conlleva a un ahorro en memoria.

En el algoritmo 1.4 se presenta la implementación de una clase la cual instancia objetos con una cantidad determinada de atributos. Fíjese como los objetos de esta clase no tienen el atributo $\_\_dict\_\_$ y, así mismo, no permite que se le agreguen otros atributos.

```
class MyClass:
    __slots__ = ['x', 'y', 'z']

myObj = MyClass()
myObj.x = 1; myObj.y = 2; myObj.z = 3
```

```
try:
    print(myObj.__dict__)
except Exception as e:
    print(e)  # 'MyClass' object has no attribute '__dict__'

try:
    myObj.w = 0
except Exception as e:
    print(e)  # 'MyClass' object has no attribute 'w'
```

Algoritmo 1.4: Instancias con una cantidad de atributos definida.

Un comentario final. Las clases que deben tener este comportamiento pero que heredan de otra clase, deben asegurarse que la *clase padre* defina la variable $\_\_slots\_\_$. Así mismo, se debe evitar que la clase hija defina variables en el $\_\_slots\_\_$ que ya han sido definidas en la clase padre[1].

Es por esto que la clase *UniqueInstances* define la variable $\_\_slots\_\_$ como una *lista* vacía.

1.3.2. `__new__` y registro

En Python, todos los objetos primero se crean y después se inicializan. El lugar donde se crean los objetos es en *metodo* $\_\_new\_\_$.

Lo dicho anteriormente se explica mejor con el siguiente ejemplo

```
class MyClass:
    def __new__(cls):
        print("__new__ method called")
        return super(MyClass, cls).__new__(cls)

myObj = MyClass()  # __new__ method called
```

Esta clase, aunque tiene más líneas que las clases de la sección anterior, a no ser por el mensaje que se imprime, es igual a

```
class MyClass:
    pass
```

Este método puede ser utilizado para definir como se inicializan los objetos. Es aquí donde se evita instanciar objetos con los mismos atributos.

En el algoritmo 1.5 se recogen las anteriores ideas. Fíjese que cuando se quiere crear un objeto con los mismos valores con los que otro objeto fue instanciado, el método $\_\_new\_\_$ no lo permite, ya que detecta que dicho método fue llamado ya con los mismos argumentos.

```
class MyClass:
    instancesAttrs = set()
```

```
    def __new__(cls, *args):
        if args not in cls.instancesAttrs:
            cls.instancesAttrs.add(args)
            print("args didn't found. Instance created")
            return super(MyClass, cls).__new__(cls)
        else:
            print("args were found. Instance wasn't created")
            return None

myObj1 = MyClass(0, 0, 0)  # args didn't found...
myObj2 = MyClass(-1, -1, -1)  # args didn't found...
myObj3 = MyClass(0, 0, 0)  # args were found...
```

Algoritmo 1.5: Instancias de una clase con valores de atributos diferentes a las demás instancias.

Un comentario final. Una vez se ejecuta el método $\_\_new\_\_$ automáticamente se ejecuta el método $\_\_init\_\_$, siempre y cuando $\_\_new\_\_$ no regrese *None*. Es decir que, del algoritmo anterior, se tienen

```
print(myObj3)  # None
```

1.3.3. __setattr__

En Python, el método $\_\_setattr\_\_$ es llamado cuando se asigna un atributo a un objeto, por ejemplo

```
class MyClass:
    def __setattr__(self, name, value):
        print("__setattr__ method called")

myObj = MyClass()
myObj.x = 0  # __setattr__ method called
```

El método $\_\_setattr\_\_$ siempre se llama, aún cuando se asignan variables al objeto dentro de la clase, es decir

```
class MySubClass(MyClass):
    def __init__(self, x):
        self.x = x  # __setattr__ method called

myObj = MySubClass(1)
```

Teniendo en cuenta lo anteriormente mostrado, en las lineas 23 a la 39 del algoritmo 1.3 se implementó el método $\_\_setattr\_\_$, donde lo primero que se verifica es si el atributo a asignar al objeto ya existe, es decir

```
        if hasattr(self, key):
```

En caso de que el atributo no haya sigo asignado al objeto, se procede normalmente, es decir

```
super(UniqueInstances, self).__setattr__(key, value)
```

En caso de que el atributo ya haya sido asignado al objeto, se crea una *tupla* con los valores de los atributos que tendría el objeto, es decir,

```
instanceAttrs = tuple((getattr(self, _key) if _key !=
↪  key else value
                       for _key in self.__slots__))
```

y se verifica que si dicha *tupla* está o no en el registro de los valores de los atributos de las instancias creadas, es decir,

```
if instanceAttrs not in
↪  self.__class__.instancesAttrs:
```

En caso de que dicha *tupla* no se encuentre en el registro, se elimina el la *tupla* del estado actual del objeto, se registra la *tupla* del estado futuro del objeto y se actualiza el valor del atributo del objeto, es decir

```
self.__class__.instancesAttrs.\
    remove(self.instanceAttrs)
self.__class__.instancesAttrs.add(instanceAttrs)

super(UniqueInstances, self).__setattr__(key, value)
```

En el caso que dicha *tupla* se encuentre en el registro, no se actualiza

```
print("Warning: " +
      "There is another instance of the class " +
      "'{}'".format(self.__class__.__name__) +
      " with the same attibutes. The object was
      ↪  not changed.")

return None
```

1.3.4. __del__

En Python, el método *__del__* es llamado cuando se pierde la referencia a un objeto, es decir

```
class MyClass:
    def __del__(self):
        print("__del__ method called")

myObj = MyClass()
myObj = None  # __del__ method called
```

Cuando esto ocurre, debemos actualizar el registro, tal como sucede en el algoritmo 1.3

```
    def __del__(self):
        self.__class__.instancesAttrs.remove(self.instanceAttrs)
```

1.3.5. Los otros métodos

Los otros métodos de la clase *UniqueInstances* son muy generales, los cuales quedan para estudio del lector.

1.3.6. Aplicaciones

Tal como se dijo anteriormente, la clase *UniqueInstances* debe ser heredada por alguna otra clase para que sea útil. Usemos esta clase para almacenar coordenadas espaciales sin que éstas se repitan. Para ello creamos una nueva clase llamada *Coordinates* que herede de la clase *UniqueInstances* y que define tres atributos, es decir

```
class Coordinate(UniqueInstances):
    __slots__ = ['x', 'y', 'z']

    def __init__(self, x, y, z):
        super(Coordinate, self).__init__(x, y, z)
```

Fíjese como asignamos a la variable $__slots__$ las cadenas de texto x, y y z, diciéndole a nuestra clase *Coordinate* que sus atributos son los que se definan en dicha variable y los que herede, que, en este caso, no hereda ningún atributo.

Fíjese también como se sobrecarga el método $__init__$, llamado al método $__init__$ de la clase padre, es decir el $__init__$ de la clase *UniqueInstances*.

Con estas líneas de código seremos capaces de lograr lo propuesto, objetos tipo *Coordinate* con valores de sus argumentos únicos.

Primero, podemos ver los argumentos con los que se han creado los objetos de dicha clase

```
    # view instances' attrs
    print(Coordinate.instancesAttrs)  # set()
```

que nos muestra un *set* vacío, ya que no se han creado objetos de dicha clase.

También podemos instanciar objetos de nuestra clase

```
    coordinate1 = Coordinate(0, 0, 0)
    print(coordinate1)  # Coordinate(0, 0, 0)
```

Fíjese que para imprimir el objeto se usó el método $__repr__$ de la clase *UniqueInstances*.

Podemos intentar agregar atributos diferentes a los que se presentan en $__slots__$ pero no lo vamos a lograr

```
try:
    coordinate1.a = 1
except Exception as e:
    print(e)  # 'Coordinate' object has no attribute 'a'
```

Fíjese que nunca especificamos dicho comportamiento. Éste viene de delegar la responsabilidad al método *__setattr__* de la clase de la cual hereda *UniqueInstances*, que, en este caso, es de la clase por defecto *type*.

Podemos intentar instanciar un objeto *Coordinate* con los mismos valores de otro objeto de la clase *Coordinate* pero no lo vamos a lograr

```
coordinate2 = eval(repr(coordinate1))  # Warning: There is
↪  another instance of the class 'Coordinate' with the same
↪  attibutes. The object was not created.
print(coordinate2)  # None
```

Fíjese que *repr* llama al método *__repr__* de la clase *UniqueInstances* y que no se puede crear el objeto de la clase *Coordinate* debido al método *__new__* detecta que ya se ha creado un objeto con los mismos argumentos.

También podemos eliminar el objeto, actualizando el registro de los valores de los atributos de los objetos existentes

```
# del first object
del(coordinate1)

# view instances' attrs
print(Coordinate.instancesAttrs)  # set()
```

Podemos instanciar más objetos, siempre y cuando se instancien con valores diferentes a los de los argumentos de los demás objetos

```
coordinate2 = Coordinate(0, 0, 0)
coordinate3 = Coordinate(1, 0, 0)
```

Ver los valores de los atributos de los objetos existentes

```
print(Coordinate.instancesAttrs)  # {(1, 0, 0), (0, 0, 0)}
```

Pero no podemos cambiar un objeto par que tome el valor de otro

```
coordinate3.x = 0  # Warning: There is another instance of
↪  the class 'Coordinate' with the same attibutes. The
↪  object was not changed.
print(coordinate3)  # Coordinate(1, 0, 0)
```

Lamentablemente, la variable *instancesAttrs* de la clase *UniqueInstances* no es diferente de la variable *instancesAttrs* de la clase *Coordinate*

```
print(UniqueInstances.instancesAttrs)  # {(1, 0, 0), (0, 0,
↪  0)}
print(Coordinate.instancesAttrs)  # {(1, 0, 0), (0, 0, 0)}
```

lo que no permite crear cuantas clases sean necesarias con variables *instancesAttrs* independientes.

2. Metaclase *UniqueInstances*

Para evitar que las clases que heredan de la clase *UniqueInstances* hagan referencia a la misma variable *instancesAttrs*, se debe crear dicha variable para cada una de las clases. Esto implica usar metaclases. En el algoritmo 2.1 se implementa la metaclase *UniqueInstances*, la cual no le permite a las clases instanciadas con dicha metaclase *a*) crear objetos con los mismos argumentos con los que otros objetos de la misma clase fueron instanciados, *b*) que los atributos de un objeto varíen para ser iguales a los de otro objeto de la misma clase y que *c*) cada clase creada tenga su propia variable *instancesAttrs*.

```
1  class UniqueInstances(type):
2      def init(self, *args):
3          for key, value in zip(self.__slots__, args):
4              setattr(self, key, value)
5
6      def setattr(self, key, value):
7          if hasattr(self, key):
8              _instanceAttr = tuple((getattr(self, _key)
9                                     if _key != key else value
10                                    for _key in self.__slots__))
11
12             if _instanceAttr not in
                ↪  self.__class__.instancesAttrs:
13                 self.__class__.instancesAttrs.\
14                     remove(self.instanceAttrs)
15                 self.__class__.instancesAttrs.add(_instanceAttr)
16             else:
17                 print("Warning: " +
18                       "There is another instance of the class " +
19                       "'{}'".format(self.__class__.__name__) +
20                       " with the same attibutes. The object was
                          ↪  not changed.")
21
22                 return None
23
24         object.__setattr__(self, key, value)
25
26     def get_instancesAttr(self):
27         return tuple(getattr(self, key) for key in
           ↪  self.__slots__)
28
29     def repr(self):
30         return "{}{}".format(self.__class__.__name__,
           ↪  str(self.instanceAttrs))
31
```

```
32      def delete(self):
33          self.__class__.instancesAttrs.remove(self.instanceAttrs)
34
35      def __new__(cls, name, bases, dict):
36          dict['instancesAttrs'] = set()
37          dict['__init__'] = UniqueInstances.init
38          dict['__setattr__'] = UniqueInstances.setattr
39          dict['instanceAttrs'] =
            ↪  property(UniqueInstances.get_instancesAttr)
40          dict['__repr__'] = UniqueInstances.repr
41          dict['__del__'] = UniqueInstances.delete
42
43          return super().__new__(cls, name, bases, dict)
44
45      def __call__(cls, *args):
46          assert len(args) == len(cls.__slots__)
47
48          if args not in cls.instancesAttrs:
49              cls.instancesAttrs.add(args)
50
51              return super().__call__(*args)
52          else:
53              print("Warning: " +
54                    "There is another instance of the class " +
55                    "'{}' ".format(cls.__name__) +
56                    "with the same attributes. The object was not
                      ↪  created.")
57
58              return None
```

Algoritmo 2.1: Metaclase *UniqueInstances* que instancia clases con un *set* como atributo a las cuales no se le permiten crear objetos con los mismos argumentos ni que estos varíen para ser iguales a otros.

Dicha metaclase hace uso de varias herramientas de la *programación orientada a objetos*, las cuales se detallan a continuación.

2.1. Metaclases

En Python, las metaclases nos permiten *interceptar* y *aumentar* la creación de clases. En consecuencia, ellas proveen un protocolo general para manejar *objetos de clase* en el programa [1]. En la practica, las metaclases permite un mayor control sobre como las clases funcionan.

Ya que las clases también son un objeto, es decir

```
class MyClass:
    pass
```

```
myObj = MyClass()

print(type(myObj))  # <class '__main__.MyClass'>

print(type(MyClass))  # <class 'type'>
```

se puede interceptar la creación de clases para controlarlas.

Dicha interceptación se hace mediante la definición de una metaclase, es decir

```
class MyMetaClass(type):
    def __new__(cls, name, bases, dict):
        print("__new__ method called")
        return super().__new__(cls, name, bases, dict)
```

Para que esta metaclase intercepte la creación de clases, hay que especificarlo en las clases de la siguiente manera

```
class MyClass(metaclass=MyMetaClass):  # __new__ method called
    pass
```

de manera que *MyClass* ya no es de tipo *type* sino de tipo *MyMetaClass*, es decir

```
print(type(MyClass))  # <class '__main__.MyMetaClass'>
```

De esta manera se puede controlar la creación de clases y, en este caso, darle métodos y atributos por defecto. Esto se hace en las líneas 50 a la 57 del algoritmo 2.1, donde se reescribieron los métodos del algoritmo 1.3, es decir

```
            return super().__call__(*args)
        else:
            print("Warning: " +
                  "There is another instance of the class " +
                  "'{}' ".format(cls.__name__) +
                  "with the same attributes. The object was not created.")
```

2.2. Aplicaciones

Todo lo dicho de la sección anterior funciona. Adicionalmente, cada una de las clases creadas a partir de la metaclase *UniqueInstances* tienen un atributo *instancesAttrs* independiente del resto, es decir

3. Conclusiones

Se ha creado una metaclase que al ser usada por otras clases hace que las instancias de dichas clases sean únicas.

Referencias

[1] Mark Lutz. *Learning Python.* O'Reilly, Sebastopol, CA, 2013.