**Introducción a la herencia múltiple y super ()**

Una guía introductoria de Pythonista a la herencia múltiple, la función super () y cómo navegar por el problema del diamante.

**Visión general rápida de la herencia**

A medida que crezca sus proyectos y paquetes de Python, inevitablemente querrá utilizar clases y aplicar el principio DRY (no repetir usted mismo) mientras lo hace. La herencia de clases es una forma fantástica de crear una clase basada en otra clase para permanecer SECO. Esta publicación cubrirá conceptos más avanzados de herencia, y la herencia básica no se tratará en profundidad.

Entonces, ¿qué es la herencia de clases? De manera similar a la genética, una clase secundaria puede "heredar" atributos y métodos de un padre. Pasemos directamente a un código para ver un ejemplo. En el siguiente bloque de código, demostraremos la herencia con una Childclase que hereda de una Parentclase.

**Aporte**

class Parent:

def \_\_init\_\_(self):

self.parent\_attribute = 'I am a parent'

def parent\_method(self):

print('Back in my day...')

# Create a child class that inherits from Parent

class Child(Parent):

def \_\_init\_\_(self):

Parent.\_\_init\_\_(self)

self.child\_attribute = 'I am a child'

# Create instance of child

child = Child()

# Show attributes and methods of child class

print(child.child\_attribute)

print(child.parent\_attribute)

child.parent\_method()

**Producción**

I am a child

I am a parent

Back in my day...

Vemos que la Childclase 'hereda' atributos y métodos de la Parentclase. Sin ningún trabajo de nuestra parte, el Parent.parent\_methodes parte de la Childclase. Para obtener los beneficios del Parent.\_\_init\_\_()método, necesitábamos llamar explícitamente al método y pasar self. Esto se debe a que cuando agregamos un \_\_init\_\_método a Child, sobrescribimos el método heredado \_\_init\_\_.

Con esa descripción breve y muy poco completa fuera del camino, saltemos al meollo del mensaje.

**Introducción a super**

En el caso más simple, la superfunción se puede usar para reemplazar la llamada explícita a Parent.\_\_init\_\_(self). Nuestro ejemplo de introducción de la primera sección se puede reescribir supercomo se ve a continuación. Tenga en cuenta que el bloque de código a continuación está escrito en Python 3, las versiones anteriores usan una sintaxis ligeramente diferente. Además, la salida se ha omitido ya que es idéntica al primer bloque de código.

class Parent:

def \_\_init\_\_(self):

self.parent\_attribute = 'I am a parent'

def parent\_method(self):

print('Back in my day...')

# Create a child class that inherits from Parent

class Child(Parent):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.child\_attribute = 'I am a parent'

# Create instance of child

child = Child()

# Show attributes and methods of child class

print(child.child\_attribute)

print(child.parent\_attribute)

child.parent\_method()

Para ser honesto, superen este caso nos da poca o ninguna ventaja. Dependiendo del nombre de nuestra clase principal, podríamos guardar algunas pulsaciones de teclas y no tenemos que pasar selfa la llamada a \_\_init\_\_. A continuación se presentan algunos pros y contras del uso de superen casos de herencia única.

**Contras**

Se puede argumentar que usar superaquí hace que el código sea menos explícito. Hacer que el código sea menos explícito viola [The Zen of Python](https://www.python.org/dev/peps/pep-0020/" \l "the-zen-of-python) , que dice: "Explícito es mejor que implícito".

**Pros**

Existe un argumento de capacidad de mantenimiento que se puede aplicar superincluso en una herencia única. Si por alguna razón su clase secundaria cambia su patrón de herencia (es decir, la clase principal cambia o hay un cambio a herencia múltiple), entonces no es necesario buscar y reemplazar todas las referencias persistentes a ParentClass.method\_name(); el uso de superpermitirá que todos los cambios fluyan con el cambio en la classdeclaración.

**super y herencia múltiple**

Antes de entrar en herencia múltiple y super... Advertencia, esto puede volverse bastante extraño y complicado.

En primer lugar, ¿qué es la herencia múltiple? Hasta ahora, el código de ejemplo ha cubierto una clase secundaria única que hereda de una clase principal única. En la herencia múltiple, hay más de una clase principal. Una clase secundaria puede heredar de 2, 3, 10, etc. clases principales.

Aquí es donde los beneficios de se supervuelven más claros. Además de ahorrar pulsaciones de teclas al hacer referencia a los diferentes nombres de las clases principales, el uso supercon múltiples patrones de herencia presenta ventajas matizadas . En resumen, si va a usar herencia múltiple, use super.

**Herencia múltiple sin super**

Veamos un ejemplo de herencia múltiple que evita modificar cualquier método padre y, a su vez, evita super.

**Aporte**

class B:

def b(self):

print('b')

class C:

def c(self):

print('c')

class D(B, C):

def d(self):

print('d')

d = D()

d.b()

d.c()

d.d()

**Producción**

b

c

d

**Orden de resolución múltiple**

Este resultado no es demasiado sorprendente dado el concepto de herencia múltiple. Dheredó los métodos xy zde sus clases padre, y todo está bien en el mundo ... por ahora.

Entonces, ¿qué si ambos By Cambos tenían un método con el mismo nombre? Aquí es donde entra en juego un concepto llamado 'orden de resolución múltiple' o MRO para abreviar. El MRO de una clase secundaria es lo que decide dónde buscará Python un método determinado y qué método se llamará cuando haya un conflicto.

Veamos un ejemplo.

**Aporte**

class B:

def x(self):

print('x: B')

class C:

def x(self):

print('x: C')

class D(B, C):

pass

d = D()

d.x()

print(D.mro())

**Producción**

x: B

[<class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.C'>, <class 'object'>]

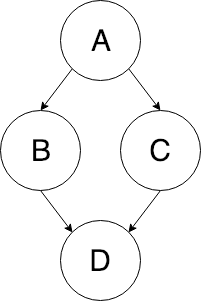
Cuando llamamos al xmétodo heredado , solo vemos la salida heredada de B. Podemos ver el MRO de nuestra Dclase llamando al mrométodo de la clase. De la D.mro()salida aprendemos lo siguiente: nuestro programa intentará llamar a Dmétodos por defecto, luego recurrirá a B, luego Cy finalmente object. Si no se encuentra en ninguno de esos lugares, obtendremos el error que Dno tiene el método que solicitamos.

Vale la pena señalar que, de forma predeterminada, todas las clases heredan objecty están al final de cada MRO.

**Herencia múltiple super, y el problema del diamante**

A continuación se muestra un ejemplo del uso de super para manejar MRO de init de una manera que sea beneficiosa. En el ejemplo, creamos una serie de clases de procesamiento de texto y combinamos su funcionalidad en otra clase con herencia múltiple. Crearemos 4 clases y la estructura de la herencia seguirá la estructura del diagrama a continuación.

*Nota: Esta estructura tiene fines ilustrativos y, salvo restricciones, habría mejores formas de implementar esta lógica.*



En realidad, este es un ejemplo del "problema del diamante" de la herencia múltiple. Por supuesto, su nombre se basa en la forma de su diseño y en el hecho de que es un problema bastante confuso.

A continuación, el diseño está escrito con el uso de super.

**Aporte**

class Tokenizer:

"""Tokenize text"""

def \_\_init\_\_(self, text):

print('Start Tokenizer.\_\_init\_\_()')

self.tokens = text.split()

print('End Tokenizer.\_\_init\_\_()')

class WordCounter(Tokenizer):

"""Count words in text"""

def \_\_init\_\_(self, text):

print('Start WordCounter.\_\_init\_\_()')

super().\_\_init\_\_(text)

self.word\_count = len(self.tokens)

print('End WordCounter.\_\_init\_\_()')

class Vocabulary(Tokenizer):

"""Find unique words in text"""

def \_\_init\_\_(self, text):

print('Start init Vocabulary.\_\_init\_\_()')

super().\_\_init\_\_(text)

self.vocab = set(self.tokens)

print('End init Vocabulary.\_\_init\_\_()')

class TextDescriber(WordCounter, Vocabulary):

"""Describe text with multiple metrics"""

def \_\_init\_\_(self, text):

print('Start init TextDescriber.\_\_init\_\_()')

super().\_\_init\_\_(text)

print('End init TextDescriber.\_\_init\_\_()')

td = TextDescriber('row row row your boat')

print('--------')

print(td.tokens)

print(td.vocab)

print(td.word\_count)

**Producción**

Start init TextDescriber.\_\_init\_\_()

Start WordCounter.\_\_init\_\_()

Start init Vocabulary.\_\_init\_\_()

Start Tokenizer.\_\_init\_\_()

End Tokenizer.\_\_init\_\_()

End init Vocabulary.\_\_init\_\_()

End WordCounter.\_\_init\_\_()

End init TextDescriber.\_\_init\_\_()

--------

['row', 'row', 'row', 'your', 'boat']

{'boat', 'your', 'row'}

5

En primer lugar, vemos que la TextDescriberclase ha heredado todos los atributos del árbol genealógico de la clase. Gracias a la herencia múltiple podemos 'combinar' la funcionalidad de más de una clase.

Analicemos ahora las impresiones que provienen de los initmétodos de la clase :

**Cada \_\_init\_\_método se llamó una vez y solo una vez.**

La TextDescriberclase heredada de 2 clases que heredan Tokenizer. ¿Por qué Tokenizer.\_\_init\_\_no se llamó dos veces?

Si reemplazamos *todas* nuestras llamadas a supercon la forma antigua, terminaríamos con 2 llamadas a Tokenizer.\_\_init\_\_. Las llamadas a super'pensar' en nuestro patrón un poco más y se salta el viaje adicional a A.

**Cada \_\_init\_\_método se inició antes de que se terminaran los demás.**

\_\_init\_\_Vale la pena señalar el orden de inicio y finalización de cada uno en caso de que esté intentando establecer un atributo que tiene un conflicto de nombres con otra clase principal. El atributo se sobrescribirá y puede resultar muy confuso.

En nuestro caso, evitamos los conflictos de nombres con los atributos heredados, por lo que todo funciona como se esperaba.

Para reiterar, el problema de los diamantes puede complicarse rápidamente y dar lugar a resultados inesperados. Con la mayoría de los casos en programación, es mejor evitar diseños complicados.

**Lo que aprendimos**

* Aprendimos sobre la superfunción y cómo se puede usar para reemplazar ParentName.methoden herencia única. Esta puede ser una práctica más fácil de mantener.
* Aprendimos sobre la herencia múltiple y cómo podemos transmitir la funcionalidad de múltiples clases principales a una sola clase secundaria.
* Aprendimos sobre el orden de resolución múltiple y cómo decide qué sucede en la herencia múltiple cuando hay un conflicto de nombres entre los métodos principales.
* Aprendimos sobre el problema del diamante y vimos un ejemplo de cómo el uso de supernavega por el diamante.