Módulo 3: Programación Orientada a Objetos - Herencia

Herencia: ¿por qué y cómo?

Antes de comenzar a hablar sobre la herencia, queremos presentar un nuevo y práctico mecanismo utilizado por las clases y los objetos de Python: es **la forma en que el objeto puede presentarse a si mismo**.

Comencemos con un ejemplo.

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


sun = Star("Sol", "Vía Láctea")
print(sun)
```

El programa imprime solo una línea de texto, que en nuestro caso es:

```
<__main__.Star object at 0x7f1074cc7c50>
```

Si ejecutas el mismo código en tu computadora, verás algo muy similar, aunque el número hexadecimal (la subcadena que comienza con 0x) será diferente, ya que es solo un identificador de objeto interno utilizado por Python, y es poco probable que aparezca igual cuando se ejecuta el mismo código en un entorno diferente.

Como puedes ver, la impresión aquí no es realmente útil, y algo más específico, es preferible.

Afortunadamente, Python ofrece tal función.

Cuando Python necesita que alguna clase u objeto deba ser presentado como una cadena (es recomendable colocar el objeto como argumento en la invocación de la función `print()`), intenta invocar un método llamado `__str__()` del objeto y emplear la cadena que devuelve.

El método por default `__str__()` devuelve la cadena anterior: fea y poco informativa. Puedes cambiarlo **definiendo tu propio método**.

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

    def __str__(self):
        return self.name + ' en ' + self.galaxy


sun = Star("Sol", "Vía Láctea")
print(sun)
```

El método nuevo `__str__()` genera una cadena que consiste en los nombres de la estrella y la galaxia, nada especial, pero los resultados de impresión se ven mejor ahora, ¿no?

El término herencia es más antiguo que la programación de computadoras, y describe la práctica común de pasar diferentes bienes de una persona a otra después de la muerte de esa persona. El término, cuando se relaciona con la programación de computadoras, tiene un significado completamente diferente.

Definamos el término para nuestros propósitos:

La herencia es una práctica común (en la programación de objetos) de **pasar atributos y métodos de la superclase (definida y existente) a una clase recién creada, llamada subclase**.

En otras palabras, la herencia **es una forma de construir una nueva clase, no desde cero, sino utilizando un repertorio de rasgos ya definido**. La nueva clase hereda (y esta es la clave) todo el equipamiento ya existente, pero puedes agregar algo nuevo si es necesario.

Gracias a eso, es posible **construir clases más especializadas (más concretas)** utilizando algunos conjuntos de reglas y comportamientos generales predefinidos.

El factor más importante del proceso es la relación entre la superclase y todas sus subclases (nota: si B es una subclase de A y C es una subclase de B, esto también significa que C es una subclase de A, ya que la relación es totalmente transitiva).

Aquí se presenta un ejemplo muy simple de **herencia de dos niveles**:

```
class Vehicle:
    pass
```

```
class LandVehicle(Vehicle):
    pass


class TrackedVehicle(LandVehicle):
    pass
```

Todas las clases presentadas están vacías por ahora, ya que te mostraremos cómo funcionan las relaciones mutuas entre las superclases y las subclases. Las llenaremos con contenido pronto.

Podemos decir que:

- La clase Vehicle es la superclase para clases LandVehicle y TrackedVehicle.
- La clase LandVehicle es una subclase de Vehicle y la superclase de TrackedVehicle al mismo tiempo.
- La clase TrackedVehicle es una subclase tanto de Vehicle y LandVehicle.

El conocimiento anterior proviene de la lectura del código (en otras palabras, lo sabemos porque podemos verlo).

¿Python sabe lo mismo? ¿Es posible preguntarle a Python al respecto? Sí lo es.

Herencia: issubclass()

Python ofrece una función que es capaz de **identificar una relación entre dos clases**, y aunque su diagnóstico no es complejo, puede **verificar si una clase particular es una subclase de cualquier otra clase**.

Así es como se ve:

```
issubclass(ClassOne, ClassTwo)
```

La función devuelve *True* si ClassOne es una subclase de ClassTwo, y *False* de lo contrario.

Vamos a verlo en acción, puede sorprenderte.

```
class Vehicle:
    pass


class LandVehicle(Vehicle):
    pass


class TrackedVehicle(LandVehicle):
    pass


for cls1 in [Vehicle, LandVehicle, TrackedVehicle]:
    for cls2 in [Vehicle, LandVehicle, TrackedVehicle]:
        print(issubclass(cls1, cls2), end="\t")
    print()
```

Hay dos bucles anidados. Su propósito es **verificar todos los pares de clases ordenadas posibles y que imprima los resultados de la verificación para determinar si el par coincide con la relación subclase-**

superclase.

Ejecuta el código. El programa produce el siguiente resultado:

```
True    False    False
True    True    False
True    True    True
```

Hagamos que el resultado sea más legible:

↓ es una subclase de →	Vehicle	LandVehicle	TrackedVehicle
Vehicle	True	False	False
LandVehicle	True	True	False
TrackedVehicle	True	True	True

Existe una observación importante que hacer: **cada clase se considera una subclase de sí misma**.

Herencia: isinstance()

Como ya sabes, **un objeto es la encarnación de una clase**. Esto significa que el objeto es como un pastel horneado usando una receta que se incluye dentro de la clase.

Esto puede generar algunos problemas.

Supongamos que tienes un pastel (por ejemplo, resultado de un argumento pasado a tu función). Deseas saber que receta se ha utilizado para prepararlo. ¿Por qué? Porque deseas saber que esperar de él, por ejemplo, si contiene nueces o no, lo cual es información crucial para ciertas personas.

Del mismo modo, puede ser crucial si el objeto tiene (o no tiene) ciertas características. En otras palabras, **si es un objeto de cierta clase o no**.

Tal hecho podría ser detectado por la función llamada `isinstance()`:

```
isinstance(objectName, ClassName)
```

La función devuelve *True* si el objeto es una instancia de la clase, o *False* de lo contrario.

Ser una instancia de una clase significa que el objeto (el pastel) se ha preparado utilizando una receta contenida en la clase o en una de sus superclases.

No lo olvides: si una subclase contiene al menos las mismas características que cualquiera de sus superclases, significa que los objetos de la subclase pueden hacer lo mismo que los objetos derivados de la superclase, por lo tanto, es una instancia de su clase de inicio y cualquiera de sus superclases.

```
class Vehicle:
    pass
```

```
class LandVehicle(Vehicle):
    pass


class TrackedVehicle(LandVehicle):
    pass


my_vehicle = Vehicle()
my_land_vehicle = LandVehicle()
my_tracked_vehicle = TrackedVehicle()

for obj in [my_vehicle, my_land_vehicle, my_tracked_vehicle]:
    for cls in [Vehicle, LandVehicle, TrackedVehicle]:
        print(isinstance(obj, cls), end="\t")
    print()
```

Hemos creado tres objetos, uno para cada una de las clases. Luego, usando dos bucles anidados, verificamos todos los pares posibles de clase de objeto para averiguar si los objetos son instancias de las clases.

Ejecuta el código.

Esto es lo que obtenemos:

```
True    False    False
True    True    False
True    True    True
```

Hagamos que el resultado sea más legible:

↓ es una instancia de →	Vehicle	LandVehicle	TrackedVehicle
my_vehicle	True	False	False
my_land_vehicle	True	True	False
my_tracked_vehicle	True	True	True

Herencia: el operador is

También existe un operador de Python que vale la pena mencionar, ya que se refiere directamente a los objetos: aquí está:

```
object_one is object_two
```

El operador *is* verifica si dos variables, en este caso (object_one y object_two) **se refieren al mismo objeto**.

No olvides que las variables no almacenan los objetos en sí, sino solo los identificadores que apuntan a la memoria interna de Python.

Asignar un valor de una variable de objeto a otra variable no copia el objeto, sino solo su identificador. Es por ello que un operador como *is* puede ser muy útil en ciertas circunstancias.

Echa un vistazo al código en el editor.

```
class SampleClass:
    def __init__(self, val):
        self.val = val


object_1 = SampleClass(0)
object_2 = SampleClass(2)
object_3 = object_1
object_3.val += 1

print(object_1 is object_2)
print(object_2 is object_3)
print(object_3 is object_1)
print(object_1.val, object_2.val, object_3.val)

string_1 = "Mary tenía un "
string_2 = "Mary tenía un corderito"
string_1 += "corderito"

print(string_1 == string_2, string_1 is string_2)
```

Analicémoslo:

- Existe una clase muy simple equipada con un constructor simple, que crea una sola propiedad. La clase se usa para instanciar dos objetos. El primero se asigna a otra variable, y su propiedad val se incrementa en uno.
- Luego, el operador is se aplica tres veces para verificar todos los pares de objetos posibles, y todos los valores de la propiedad val son mostrados en pantalla.
- La última parte del código lleva a cabo otro experimento. Después de tres tareas, ambas cadenas contienen los mismos textos, pero estos textos se almacenan en diferentes objetos.

El código imprime:

```
False
False
True
1 2 1
True False
```

Los resultados prueban que *object_1* y *object_3* son en realidad los mismos objetos, mientras que *string_1* y *string_2* no lo son, a pesar de que su contenido sea el mismo.

Cómo Python encuentra propiedades y métodos

Ahora veremos como Python trata con los métodos de herencia.

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

    def __str__(self):
```

```
        return "Mi nombre es " + self.name + "."


class Sub(Super):
    def __init__(self, name):
        Super.__init__(self, name)


obj = Sub("Andy")

print(obj)
```

Vamos a analizarlo:

- Existe una clase llamada *Super*, que define su propio constructor utilizado para asignar la propiedad del objeto, llamada *name*.
- La clase también define el método `__str__()`, lo que permite que la clase pueda presentar su identidad en forma de texto.
- La clase se usa luego como base para crear una subclase llamada *Sub*. La clase *Sub* define su propio constructor, que invoca el de la superclase. Toma nota de como lo hemos hecho: *Super.__init__(self, name)*.
- Hemos nombrado explícitamente la superclase y hemos apuntado al método para invocar a `__init__()`, proporcionando todos los argumentos necesarios.
- Hemos instanciado un objeto de la clase Sub y lo hemos impreso.

El código da como salida:

```
Mi nombre es Andy.
```

Nota: Como no existe el método `__str__()` dentro de la clase *Sub*, la cadena a imprimir se producirá dentro de la clase Super. Esto significa que el método `__str__()` ha sido heredado por la clase *Sub*.

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

    def __str__(self):
        return "Mi nombre es " + self.name + "."


class Sub(Super):
    def __init__(self, name):
        super().__init__(name)


obj = Sub("Andy")

print(obj)
```

Lo hemos modificado para mostrarte otro método de acceso a cualquier entidad definida dentro de la superclase.

En el ejemplo anterior, nombramos explícitamente la superclase. En este ejemplo, hacemos uso de la función super(), la cual **accede a la superclase sin necesidad de conocer su nombre**:

```
super().__init__(name)
```

La función `super()` crea un contexto en el que no tiene que (además, no debe) pasar el argumento propio al método que se invoca; es por eso que es posible activar el constructor de la superclase utilizando solo un argumento.

Nota: puedes usar este mecanismo no solo para **invocar al constructor de la superclase, pero también para obtener acceso a cualquiera de los recursos disponibles dentro de la superclase**.

Intentemos hacer algo similar, pero con propiedades (más precisamente con: **variables de clase**).

```
# Probando propiedades: variables de clase.
class Super:
    supVar = 1


class Sub(Super):
    subVar = 2


obj = Sub()

print(obj.subVar)
print(obj.supVar)
```

Como puedes observar, la clase *Super* define una variable de clase llamada *supVar*, y la clase *Sub* define una variable llamada *subVar*.

Ambas variables son visibles dentro del objeto de clase Sub, es por ello que el código da como salida:

```
2
1
```

El mismo efecto se puede observar con **variables de instancia**, observa el segundo ejemplo en el editor.

```
# Probando propiedades: variables de instancia.
class Super:
    def __init__(self):
        self.supVar = 11


class Sub(Super):
    def __init__(self):
        super().__init__()
        self.subVar = 12


obj = Sub()

print(obj.subVar)
print(obj.supVar)
```

El constructor de la clase *Sub* crea una variable de instancia llamada *subVar*, mientras que el constructor de *Super* hace lo mismo con una variable de nombre *supVar*. Al igual que el ejemplo anterior, ambas variables son accesibles desde el objeto de clase *Sub*.

La salida del programa es:

```
12
11
```

Nota: La existencia de la variable *supVar* obviamente está condicionada por la invocación del constructor de la clase *Super*. Omitirlo daría como resultado la ausencia de la variable en el objeto creado (pruébalo tu mismo).

Ahora es posible formular una declaración general que describa el comportamiento de Python.

Cuando intentes acceder a una entidad de cualquier objeto, Python intentará (en este orden):

- Encontrarla **dentro del objeto** mismo.
- Encontrarla **en todas las clases** involucradas en la línea de herencia del objeto de abajo hacia arriba.

Si ambos intentos fallan, una excepción (*AttributeError*) **será generada**.

La primera condición puede necesitar atención adicional. Como sabes, todos los objetos derivados de una clase en particular pueden tener diferentes conjuntos de atributos, y algunos de los atributos pueden agregarse al objeto mucho tiempo después de la creación del objeto.

El ejemplo en el editor resume esto en una **línea de herencia de tres niveles**.

```
class Level1:
    variable_1 = 100
    def __init__(self):
        self.var_1 = 101

    def fun_1(self):
        return 102


class Level2(Level1):
    variable_2 = 200
    def __init__(self):
        super().__init__()
        self.var_2 = 201

    def fun_2(self):
        return 202


class Level3(Level2):
    variable_3 = 300
    def __init__(self):
        super().__init__()
        self.var_3 = 301

    def fun_3(self):
        return 302


obj = Level3()

print(obj.variable_1, obj.var_1, obj.fun_1())
print(obj.variable_2, obj.var_2, obj.fun_2())
print(obj.variable_3, obj.var_3, obj.fun_3())
```

```
100 101 102
```

```
200 201 202
300 301 302
```

Todos los comentarios que hemos hecho hasta ahora están relacionados con **casos de herencia única**, cuando una subclase tiene exactamente una superclase. Esta es la situación más común (y también la recomendada).

Python, sin embargo, ofrece mucho más aquí. En las próximas lecciones te mostraremos algunos ejemplos de **herencia múltiple**.

La herencia múltiple ocurre cuando una clase tiene más de una superclase.

Sintácticamente, dicha herencia se presenta como una lista de superclases separadas por comas entre paréntesis después del nombre de la nueva clase, al igual que aquí:

```
class SuperA:
    var_a = 10
    def fun_a(self):
        return 11


class SuperB:
    var_b = 20
    def fun_b(self):
        return 21


class Sub(SuperA, SuperB):
    pass

obj = Sub()

print(obj.var_a, obj.fun_a())
print(obj.var_b, obj.fun_b())
```

La clase *Sub* tiene dos superclases: *SuperA* y *SuperB*. Esto significa que la clase *Sub* **hereda todos los bienes ofrecidos por ambas clases** *SuperA* y *SuperB*.

El código imprime:

```
10 11
20 21
```

Ahora es el momento de introducir un nuevo término - **overriding (anulación)**.

¿Qué crees que sucederá si más de una de las superclases define una entidad con un nombre en particular?

```
class Level1:
    var = 100
    def fun(self):
        return 101


class Level2(Level1):
    var = 200
```

```
    def fun(self):
        return 201


class Level3(Level2):
    pass


obj = Level3()

print(obj.var, obj.fun())
```

Tanto la clase, *Level1* y *Level2* definen un método llamado `fun()` y una propiedad llamada *var*. ¿Significará esto el objeto de la claseLevel3 podrá acceder a dos copias de cada entidad? De ningún modo.

La entidad definida después (en el sentido de herencia) anula la misma entidad definida anteriormente. Es por eso que el código produce el siguiente resultado:

```
200 201
```

Como puedes ver, la variable de clase *var* y el método `fun()` de la clase *Level2* anula las entidades de los mismos nombres derivados de la clase *Level1*.

Esta característica se puede usar intencionalmente para modificar el comportamiento predeterminado de las clases (o definido previamente) cuando cualquiera de tus clases necesite actuar de manera diferente a su ancestro.

¿Qué ocurre cuando una clase tiene dos ancestros que ofrecen la misma entidad y se encuentran en el mismo nivel? En otras palabras, ¿Qué se debe esperar cuando surge una clase usando herencia múltiple? Miremos lo siguiente.

```
class Left:
    var = "L"
    var_left = "LL"
    def fun(self):
        return "Left"


class Right:
    var = "R"
    var_right = "RR"
    def fun(self):
        return "Right"


class Sub(Left, Right):
    pass


obj = Sub()

print(obj.var, obj.var_left, obj.var_right, obj.fun())
```

La clase *Sub* hereda todos los bienes de dos superclases *Left* y *Right* [1)]

[1)]
estos nombres están destinados a ser significativos). No hay duda de que la variable de clase *var_right* proviene

de la clase *Right*, y *var_left* proviene de la clase *Left* respectivamente. Esto es claro. Pero, ¿De donde proviene la variable *var*? ¿Es posible adivinarlo? El mismo problema se encuentra con el método `fun()` - ¿Será invocada desde *Left* o desde *Right*? Ejecutemos el programa: la salida será:

```
L LL RR Left
```

Esto prueba que ambos casos poco claros tienen una solución dentro de la clase *Left*. ¿Es esta una premisa suficiente para formular una regla general? Sí lo es. Podemos decir que **Python busca componentes de objetos** en el siguiente orden:

- Dentro del objeto mismo.
- En sus superclases, de abajo hacia arriba.
- Si hay más de una clase en una ruta de herencia, Python las escanea de izquierda a derecha.

¿Necesitas algo más? Simplemente haz una pequeña enmienda en el código, reemplaza:class Sub(Left, Right): con: class Sub(Right, Left):, luego ejecuta el programa nuevamente y observa qué sucede. ¿Qué ves ahora? Vemos:

```
R LL RR Right
```

=== Cómo construir una jerarquía de clases Construir una jerarquía de clases no es solo por amor al arte. Si divides un problema entre las clases y decides cual de ellas debe ubicarse en la parte superior y cual debe ubicarse en la parte inferior de la jerarquía, debes analizar cuidadosamente el problema, pero antes de mostrarte como hacerlo (y como no hacerlo), queremos resaltar un efecto interesante. No es nada extraordinario (es solo una consecuencia de las reglas generales presentadas anteriormente), pero recordarlo puede ser clave para comprender como funcionan algunos códigos y cómo se puede usar este efecto para construir un conjunto flexible de clases.

```
class One:
    def do_it(self):
        print("do_it de One")

    def doanything(self):
        self.do_it()


class Two(One):
    def do_it(self):
        print("do_it de Two")


one = One()
two = Two()

one.doanything()
two.doanything()
```

Analicémoslo:

- Existen dos clases llamadas *One* y *Two*, se entiende que *Two* es derivada de *One*. Nada especial. Sin embargo, algo es notable: el `método do_it()`.
- El método `do_it()` está **definido dos veces**: originalmente dentro de *One* posteriormente dentro de *Two*. La esencia del ejemplo radica en el hecho de que es **invocado solo una vez** dentro de One.

La pregunta es: ¿cuál de los dos métodos será invocado por las dos últimas líneas del código? La primera invocación parece ser simple, el invocar el método `doanything()` del objeto llamado *one* obviamente activará el primero de los métodos. La segunda invocación necesita algo de atención. También es simple si tienes en

cuenta cómo Python encuentra los componentes de la clase. La segunda invocación ejecutará el método `do_it()` en la forma existente dentro de la clase *Two*, independientemente del hecho de que la invocación se lleva a cabo dentro de la clase One. En efecto, el código genera el siguiente resultado:

```
do_it from One
do_it from Two
```

Nota: la situación en la cual **la subclase puede modificar el comportamiento de su superclase (como en el ejemplo) se llama poliformismo**. La palabra proviene del griego (polys: «muchos, mucho» y morphe, «forma, forma»), lo que significa que una misma clase puede tomar varias formas dependiendo de las redefiniciones realizadas por cualquiera de sus subclases. El método, redefinido en cualquiera de las superclases, que cambia el comportamiento de la superclase, se llama **virtual**. En otras palabras, ninguna clase se da por hecho. El comportamiento de cada clase puede ser modificado en cualquier momento por cualquiera de sus subclases. Te mostraremos **como usar el poliformismo para extender la flexibilidad de la clase**.

```
import time

class TrackedVehicle:
    def control_track(left, stop):
        pass

    def turn(left):
        control_track(left, True)
        time.sleep(0.25)
        control_track(left, False)


class WheeledVehicle:
    def turn_front_wheels(left, on):
        pass

    def turn(left):
        turn_front_wheels(left, True)
        time.sleep(0.25)
        turn_front_wheels(left, False)
```

¿Se parece a algo? Sí, por supuesto que lo hace. Se refiere al ejemplo que se muestra al comienzo del módulo cuando hablamos de los conceptos generales de la programación orientada a objetos. Puede parecer extraño, pero no utilizamos herencia en este ejemplo, solo queríamos mostrarte que no nos limita. Definimos dos clases separadas capaces de producir dos tipos diferentes de vehículos terrestres. La principal diferencia entre ellos está en cómo giran. Un vehículo con ruedas solo gira las ruedas delanteras (generalmente). Un vehículo oruga tiene que detener una de las pistas. ¿Puedes seguir el código?

- Un vehículo oruga realiza un giro deteniéndose y moviéndose en una de sus pistas (esto lo hace el método control_track() el cual se implementará más tarde).
- Un vehículo con ruedas gira cuando sus ruedas delanteras giran (esto lo hace el método turn_front_wheels()).
- El método turn() utiliza el método adecuado para cada vehículo en particular.

¿Puedes detectar **el error del código**? Los métodos `turn()` son muy similares como para dejarlos en esta forma. Vamos a reconstruir el código: vamos a presentar una superclase para reunir todos los aspectos similares de los vehículos, trasladando todos los detalles a las subclases.

```
import time

class Vehicle:
    def change_direction(left, on):
        pass
```

```
    def turn(left):
        change_direction(left, True)
        time.sleep(0.25)
        change_direction(left, False)


class TrackedVehicle(Vehicle):
    def control_track(left, stop):
        pass

    def change_direction(left, on):
        control_track(left, on)


class WheeledVehicle(Vehicle):
    def turn_front_wheels(left, on):
        pass

    def change_direction(left, on):
        turn_front_wheels(left, on)
```

Esto es lo que hemos hecho:

- Definimos una superclase llamada *Vehicle*, la cual utiliza el método `turn()` para implementar un esquema para poder girar, mientras que el giro en si es realizado por `change_direction()`; nota: dicho método está vacío, ya que vamos a poner todos los detalles en la subclase (dicho método a menudo se denomina **método abstracto**, ya que solo demuestra alguna posibilidad que será instanciada más tarde).
- Definimos una subclase llamada *TrackedVehicle* (nota: es derivada de la clase *Vehicle*) la cual instancia el método `change_direction()` utilizando el método denominado `control_track()`.
- Respectivamente, la subclase llamada *WheeledVehicle* hace lo mismo, pero usa el método `turn_front_wheels()` para obligar al vehículo a girar.

La ventaja más importante (omitiendo los problemas de legibilidad) es que esta forma de código te permite implementar un nuevo algoritmo de giro simplemente modificando el método turn(), lo cual se puede hacer en un solo lugar, ya que todos los vehículos lo obedecerán. Así es como el **el poliformismo ayuda al desarrollador a mantener el código limpio y consistente**. La herencia no es la única forma de construir clases adaptables. Puedes lograr los mismos objetivos (no siempre, pero muy a menudo) utilizando una técnica llamada composición. **La composición es el proceso de componer un objeto usando otros objetos diferentes**. Los objetos utilizados en la composición entregan un conjunto de rasgos deseados (propiedades y/o métodos), podemos decir que actúan como bloques utilizados para construir una estructura más complicada. Puede decirse que:

- **La herencia extiende las capacidades de una clase** agregando nuevos componentes y modificando los existentes; en otras palabras, la receta completa está contenida dentro de la clase misma y todos sus ancestros; el objeto toma todas las pertenencias de la clase y las usa.
- **La composición proyecta una clase como contenedor** capaz de almacenar y usar otros objetos (derivados de otras clases) donde cada uno de los objetos implementa una parte del comportamiento de una clase.

Permítenos ilustrar la diferencia usando los vehículos previamente definidos. El enfoque anterior nos condujo a una jerarquía de clases en la que la clase más alta conocía las reglas generales utilizadas para girar el vehículo, pero no sabía cómo controlar los componentes apropiados (ruedas o pistas). Las subclases implementaron esta capacidad mediante la introducción de mecanismos especializados. Hagamos (casi) lo mismo, pero usando composición. La clase, como en el ejemplo anterior, sabe cómo girar el vehículo, pero el giro real lo realiza un

objeto especializado almacenado en una propiedad llamada *controlador*. El *controlador* es capaz de controlar el vehículo manipulando las partes relevantes del vehículo.

```
import time

class Tracks:
    def change_direction(self, left, on):
        print("pistas: ", left, on)


class Wheels:
    def change_direction(self, left, on):
        print("ruedas: ", left, on)


class Vehicle:
    def __init__(self, controller):
        self.controller = controller

    def turn(self, left):
        self.controller.change_direction(left, True)
        time.sleep(0.25)
        self.controller.change_direction(left, False)


wheeled = Vehicle(Wheels())
tracked = Vehicle(Tracks())

wheeled.turn(True)
tracked.turn(False)
```

Existen dos clases llamadas *Tracks* y *Wheels*, ellas saben como controlar la dirección del vehículo. También hay una clase llamada *Vehicle* que puede usar cualquiera de los controladores disponibles (los dos ya definidos o cualquier otro definido en el futuro): el *controlador* se pasa a la clase durante la inicialización. De esta manera, la capacidad de giro del vehículo se compone de un objeto externo, no implementado dentro de la clase *Vehicle*. En otras palabras, tenemos un vehículo universal y podemos instalar pistas o ruedas en él. El código produce el siguiente resultado:

```
ruedas:  True True
pistas:  True False
tracks:  False True
tracks:  False False
```

=== Herencia simple frente a herencia múltiple Como ya sabes, no hay obstáculos para usar la herencia múltiple en Python. Puedes derivar cualquier clase nueva de más de una clase definida previamente. Solo hay un «pero». El hecho de que puedas hacerlo no significa que tengas que hacerlo. No olvides que:

- Una sola clase de herencia siempre es más simple, segura y fácil de entender y mantener.
- La herencia múltiple siempre es arriesgada, ya que tienes muchas más oportunidades de cometer un error al identificar estas partes de las superclases que influirán efectivamente en la nueva clase.
- La herencia múltiple puede hacer que la anulación sea extremadamente difícil; además, el emplear la función super() se vuelve ambiguo.
- La herencia múltiple viola el **principio de responsabilidad única** (mas detalles aquí: https://en.wikipedia.org/wiki/Single_responsibility_principle) ya que forma una nueva clase de dos (o más) clases que no saben nada una de la otra.
- Sugerimos encarecidamente la herencia múltiple como la última de todas las posibles soluciones: si realmente necesitas las diferentes funcionalidades que ofrecen las diferentes clases, la composición

puede ser una mejor alternativa.

=== ¿Qué es el Orden de Resolución de Métodos (MRO) y por qué no todas las herencias tienen sentido? MRO, en general, es una forma (puedes llamarlo una **estrategia**) en la que un lenguaje de programación en particular escanea la parte superior de la jerarquía de una clase para encontrar el método que necesita actualmente. Vale la pena enfatizar que los diferentes lenguajes usan MROs levemente (o incluso completamente) diferentes. Python es único en este aspecto y sus costumbres son un poco específicas. Te mostraremos cómo funciona el MRO de Python en dos casos peculiares que son ejemplos claros de problemas que pueden ocurrir cuando intentas usar la herencia múltiple de manera demasiado imprudente. Comencemos con un fragmento que inicialmente puede parecer simple.

```
class Top:
    def m_top(self):
        print("top")


class Middle(Top):
    def m_middle(self):
        print("middle")


class Bottom(Middle):
    def m_bottom(self):
        print("bottom")


object = Bottom()
object.m_bottom()
object.m_middle()
object.m_top()
```

Estamos seguros de que si analizas el fragmento tu mismo, no verás ninguna anomalía en él. Sí, tienes toda la razón: parece claro y simple, y no genera preocupaciones. Si ejecutas el código, producirá el siguiente resultado predecible:

```
bottom
middle
top
```

Sin sorpresas hasta ahora. Hagamos un pequeño cambio en este código. Echa un vistazo:

```
class Top:
    def m_top(self):
        print("top")


class Middle(Top):
    def m_middle(self):
        print("middle")


class Bottom(Middle, Top):
    def m_bottom(self):
        print("bottom")


```

```
object = Bottom()
object.m_bottom()
object.m_middle()
object.m_top()
```

¿Puedes ver la diferencia? Está escondida en esta línea:

```
class Bottom(Middle, Top):
```

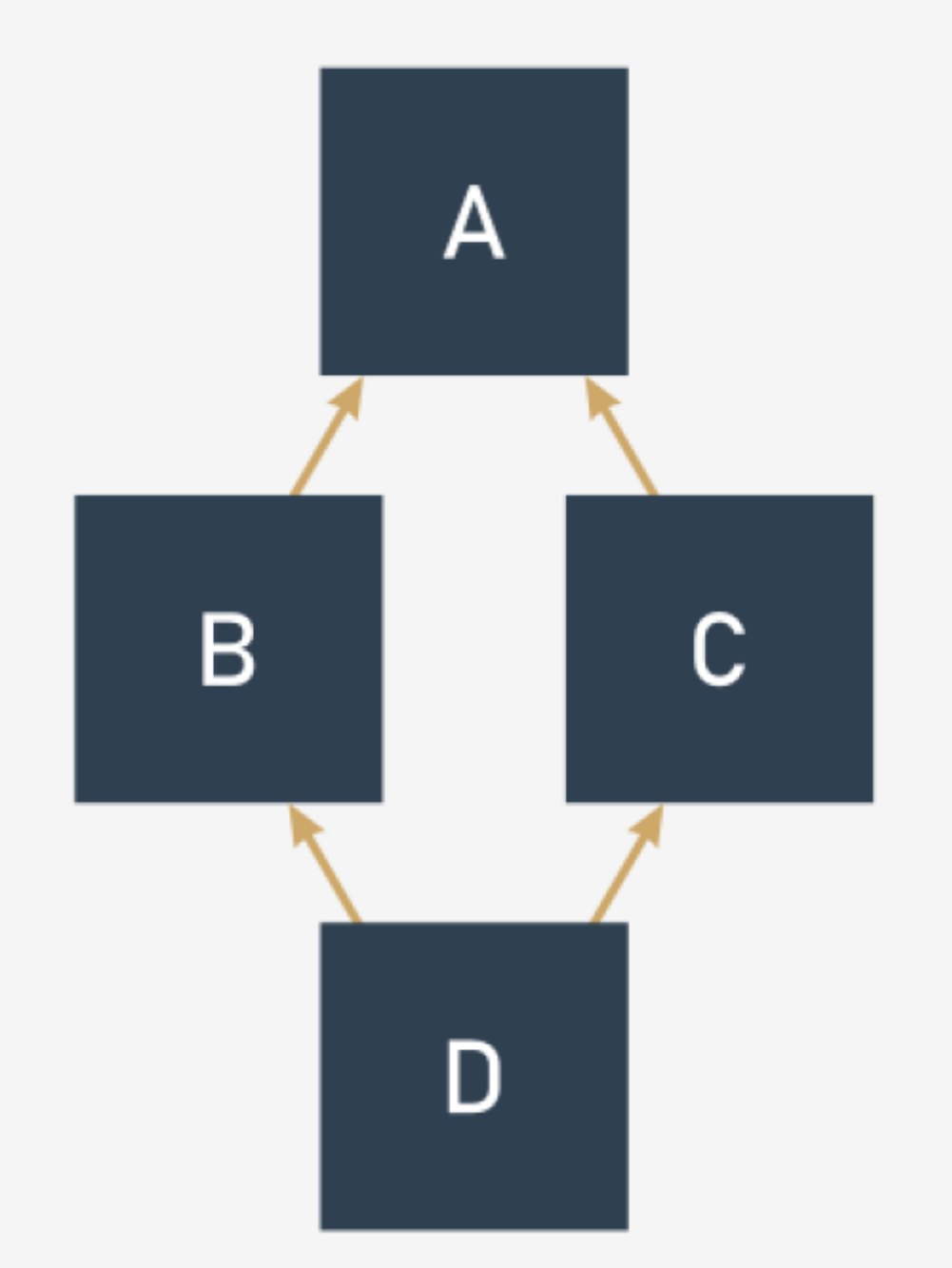
De esta manera exótica, hemos convertido un código muy simple con una clara ruta de herencia única en un misterioso acertijo de herencia múltiple. «¿Es válido?» Te puedes preguntar. Sí lo es. «¿Cómo es eso posible?» te preguntas, esperamos que realmente sientas la necesidad de hacer esta pregunta. Como puedes ver, el orden en el que se enumeran las dos superclases entre paréntesis cumple con la estructura del código: la clase *Middle* precede a la clase *Top*, justo como en la ruta de herencia real. A pesar de su rareza, la muestra es correcta y funciona como se esperaba, pero debe indicarse que esta notación no aporta ninguna funcionalidad nueva ni significado adicional. Modifiquemos el código una vez más; ahora intercambiaremos ambos nombres de superclase en la definición de clase Bottom. Así es como se ve el fragmento de código ahora:

```
class Top:
    def m_top(self):
        print("top")


class Middle(Top):
    def m_middle(self):
        print("middle")


class Bottom(Top, Middle):
    def m_bottom(self):
        print("bottom")


object = Bottom()
object.m_bottom()
object.m_middle()
object.m_top()
```

Para anticiparnos a tu pregunta, diremos que esta enmienda ha estropeado el código y ya no se ejecutará. Qué pena. El orden que intentamos forzar (Top, Middle) es incompatible con la ruta de herencia que se deriva de la estructura del código. A Python no le gustará. Esto es lo que veremos:

```
TypeError: Cannot create a consistent method resolution order (MRO) for bases Top,
Middle
```

Creemos que el mensaje habla por sí solo. El MRO de Python no se puede doblar ni violar, no solo porque esa es la forma en que funciona Python, sino también porque es una regla que debes obedecer. === El Problema del Diamante El segundo ejemplo del espectro de problemas que posiblemente pueden surgir de la herencia múltiple está ilustrado por un problema clásico llamado **problema del diamante**. El nombre refleja la forma del diagrama de herencia; observa la imagen:



- Existe la superclase superior llamada A.
- Existen dos subclases derivadas de A: B y C.
- También está la subclase inferior llamada D, derivada de B y C (o C y B, ya que estas dos variantes significan cosas diferentes en Python).

¿Puedes ver el diamante ahí?

```
class A:
    pass


class B(A):
    pass


class C(A):
    pass
```

```
class D(B, C):
    pass


d = D()
```

La misma estructura, pero expresada en Python. Algunos lenguajes de programación no permiten la herencia múltiple en absoluto y, como consecuencia, no te permitirán construir un diamante; este es el camino que Java y C# han elegido seguir desde sus orígenes. Python, sin embargo, ha elegido una ruta diferente: permite la herencia múltiple y no le importa si escribe y ejecuta código como el del editor. Pero no te olvides del MRO: siempre está a cargo. Reconstruyamos nuestro ejemplo de la página anterior para hacerlo más parecido a un diamante, como se muestra a continuación:

```
class Top:
    def m_top(self):
        print("top")


class Middle_Left(Top):
    def m_middle(self):
        print("middle_left")


class Middle_Right(Top):
    def m_middle(self):
        print("middle_right")


class Bottom(Middle_Left, Middle_Right):
    def m_bottom(self):
        print("bottom")


object = Bottom()
object.m_bottom()
object.m_middle()
object.m_top()
```

Nota: ambas clases *Middle* definen **un método con el mismo nombre**: `m_middle()`. Introduce una pequeña incertidumbre en nuestra muestra, aunque estamos absolutamente seguros de que puedes responder la siguiente pregunta clave: ¿cuál de los dos métodos `m_middle()` se invocará realmente cuando la siguiente línea se ejecute?

```
Object.m_middle()
```

En otras palabras, qué verás en la pantalla: *middle_left* o *middle_right*? No es necesario que te apresures, ¡piénselo dos veces y toma en cuenta el MRO de Python! ¿Estás listo? Sí, tienes razón. La invocación activará el método `m_middle()`, que proviene de la clase *Middle_Left*. La explicación es simple: la clase aparece antes de *Middle_Right* en la lista de herencia de la clase *Bottom*. Si deseas asegurarte de que no haya dudas al respecto, intenta intercambiar estas dos clases en la lista y verifica los resultados. Si deseas experimentar algunas impresiones más profundas sobre la herencia múltiple y las piedras preciosas, intenta modificar nuestro fragmento y equipar la clase *Upper* con otro espécimen del método `m_middle ()` e investigua su comportamiento detenidamente. Como puedes ver, los diamantes pueden traer algunos problemas a tu vida, tanto los reales como los que ofrece Python.