Creado por:

Isabel Maniega

POO (Programación Orientada a Objetos)

Objeto

Una clase (entre otras definiciones) es un conjunto de objetos. Un objeto es un ser perteneciente a una clase.

Un objeto es una encarnación de los requisitos, rasgos y cualidades asignados a una clase específica. Esto puede sonar simple, pero ten en cuenta las siguientes circunstancias importantes. Las clases forman una jerarquía.

Herencia

Cualquier objeto vinculado a un nivel específico de una jerarquía de clases hereda todos los rasgos (así como los requisitos y cualidades) definidos dentro de cualquiera de las superclases.

Objeto

La programación orientada a objetos supone que cada objeto existente puede estar equipado con tres grupos de atributos:

- Un objeto tiene un nombre que lo identifica de forma exclusiva dentro de su namespace (aunque también puede haber algunos objetos anónimos).
- Un objeto tiene un conjunto de propiedades individuales que lo hacen original, único o sobresaliente (aunque es posible que algunos objetos no tengan propiedades).
- Un objeto tiene un conjunto de habilidades para realizar actividades específicas, capaz de cambiar el objeto en sí, o algunos de los otros objetos.

In [1]:
```
# Definir una clase
# usar la palabra reservada class + nombre_clase

class TheSimplestClass:
    pass
```

In [2]:
```
# crear un objeto de esa clase

my_first_object = TheSimplestClass()
```

Nota:

- El nombre de la clase intenta fingir que es una función, ¿puedes ver esto? Lo discutiremos pronto.
- El objeto recién creado está equipado con todo lo que trae la clase. Como esta clase está completamente vacía, el objeto también está vacío.

El acto de crear un objeto de la clase seleccionada también se llama **instanciación** (ya que el objeto se convierte en una instancia de la clase).

SELF

Un método es una función que está dentro de una clase.

Existe un requisito fundamental: un método está obligado a tener al menos un parámetro (no existen métodos sin parámetros; un método puede invocarse sin un argumento, pero no puede declararse sin parámetros).

El primer (o único) parámetro generalmente se denomina self. Te sugerimos que lo sigas nombrando de esta manera, darle otros nombres puede causar sorpresas inesperadas.

El nombre self sugiere el propósito del parámetro: identifica el objeto para el cual se invoca el método.

Si deseas que el método acepte parámetros distintos a self, debes:

Colocarlos después de self en la definición del método. Pasarlos como argumentos durante la invocación sin especificar self.

In [1]:
```
class Classy:
    def method(self, par):
        print("método:", par)


obj = Classy()
obj.method(1)
obj.method(2)
obj.method(3)
```

```
método: 1
método: 2
método: 3
```

In [2]:
```
class Classy:
    def method(self):
        print("método")


obj = Classy()
obj.method()
```

```
método
```

El parámetro self es usado para obtener acceso a la instancia del objeto y las variables de clase.

In [3]:
```
class Classy:
    varia = 2
    def method(self):
        print(self.varia, self.var)


obj = Classy()
obj.var = 3
obj.method()
```

```
2 3
```

El parámetro self también se usa para invocar otros métodos desde dentro de la clase.

In [4]:
```
class Classy:
    def other(self):
        print("otro")

    def method(self):
        print("método")
        self.other()


obj = Classy()
obj.method()
```

```
método
otro
```

INIT

Si se nombra un método de esta manera: __ init __, no será un método regular, será un constructor.

Si una clase tiene un constructor, este se invoca automática e implícitamente cuando se instancia el objeto de la clase.

El constructor:

- Esta obligado a tener el parámetro self (se configura automáticamente).
- Pudiera (pero no necesariamente) tener mas parámetros que solo self; si esto sucede, la forma en que se usa el nombre de la clase para crear el objeto debe tener la definición __ init __.
- Se puede utilizar para configurar el objeto, es decir, inicializa adecuadamente su estado interno, crea variables de instancia, crea instancias de cualquier otro objeto si es necesario, etc.

Ten en cuenta que el constructor:

- No puede retornar un valor, ya que está diseñado para devolver un objeto recién creado y nada más.
- No se puede invocar directamente desde el objeto o desde dentro de la clase (puedes invocar un constructor desde cualquiera de las superclases del objeto, pero discutiremos esto más adelante).

In [5]:
```
class Classy:
    def __init__(self, value):
        self.var = value


obj_1 = Classy("objeto")

print(obj_1.var)
```

```
objeto
```

In [6]:
```
class Classy:
    def visible(self):
        print("visible")

    def __hidden(self): # Método privado
        print("oculto")


obj = Classy()
obj.visible()

try:
    obj.__hidden() # no puede acceder sin especificar la clase
except:
    print("fallido")

obj._Classy__hidden() # accede especificando la clase
```

```
visible
fallido
oculto
```

DICT

Encuentra todos los métodos y atributos definidos. Localiza el contexto en el que existen: dentro del objeto o dentro de la clase.

In [7]:
```
class Classy:
    varia = 1
    def __init__(self):
        self.var = 2

    def method(self):
        pass

    def __hidden(self):
        pass


obj = Classy()

print(obj.__dict__)
print(Classy.__dict__)
```

```
{'var': 2}
{'__module__': '__main__', 'varia': 1, '__init__': <function Classy.__init
__ at 0x7fae985a7430>, 'method': <function Classy.method at 0x7fae985a74c0
>, '_Classy__hidden': <function Classy.__hidden at 0x7fae985a7550>, '__dic
t__': <attribute '__dict__' of 'Classy' objects>, '__weakref__': <attribut
e '__weakref__' of 'Classy' objects>, '__doc__': None}
```

__ dict __ es un diccionario. Otra propiedad incorporada que vale la pena mencionar es una cadena llamada __ name __.

La propiedad contiene el nombre de la clase. No es nada emocionante, es solo una cadena.

Nota: el atributo __ name __ está ausente del objeto, existe solo dentro de las clases.

Si deseas encontrar la clase de un objeto en particular, puedes usar una función llamada type(), la cual es capaz (entre otras cosas) de encontrar una clase que se haya utilizado para crear instancias de cualquier objeto.

In [8]:
```
class Classy:
    pass


print(Classy.__name__)
obj = Classy()
print(type(obj).__name__)
```

```
Classy
Classy
```

__ module __ es una cadena, también almacena el nombre del módulo que contiene la definición de la clase.

Como sabes, cualquier módulo llamado __ main __ en realidad no es un módulo, sino es el archivo actualmente en ejecución.

In [9]:
```
class Classy:
    pass


print(Classy.__module__)
obj = Classy()
print(obj.__module__)
```

```
__main__
__main__
```

__ bases __ es una tupla. La tupla contiene clases (no nombres de clases) que son superclases directas de la clase.

El orden es el mismo que el utilizado dentro de la definición de clase.

Te mostraremos solo un ejemplo muy básico, ya que queremos resaltar cómo funciona la herencia.

Además, te mostraremos cómo usar este atributo cuando discutamos los aspectos orientados a objetos de las excepciones.

Nota: solo las clases tienen este atributo, los objetos no.

Hemos definido una función llamada printBases(), diseñada para presentar claramente el contenido de la tupla.

In [10]:
```
class SuperOne:
    pass


class SuperTwo:
    pass


class Sub(SuperOne, SuperTwo):
    pass


def printBases(cls):
    print('( ', end='')

    for x in cls.__bases__:
        print(x.__name__, end=' ')
    print(')')


printBases(SuperOne)
printBases(SuperTwo)
printBases(Sub)
```

```
( object )
( object )
( SuperOne SuperTwo )
```

Reflexión e introspección

Todo esto permite que el programador de Python realice dos actividades importantes específicas para muchos lenguajes objetivos. Las cuales son:

- Introspección, que es la capacidad de un programa para examinar el tipo o las propiedades de un objeto en tiempo de ejecución.
- Reflexión, que va un paso más allá, y es la capacidad de un programa para manipular los valores, propiedades y/o funciones de un objeto en tiempo de ejecución.

En otras palabras, no tienes que conocer la definición completa de clase/objeto para manipular el objeto, ya que el objeto y/o su clase contienen los metadatos que te permiten reconocer sus características durante la ejecución del programa.

In [11]:
```
class MyClass:
    pass


obj = MyClass()
obj.a = 1
obj.b = 2
obj.i = 3
obj.ireal = 3.5
obj.integer = 4
obj.z = 5


def incIntsI(obj):
    for name in obj.__dict__.keys():
        if name.startswith('i'):
            val = getattr(obj, name)
            if isinstance(val, int):
                setattr(obj, name, val + 1)


print(obj.__dict__)
incIntsI(obj)
print(obj.__dict__)
```

```
{'a': 1, 'b': 2, 'i': 3, 'ireal': 3.5, 'integer': 4, 'z': 5}
{'a': 1, 'b': 2, 'i': 4, 'ireal': 3.5, 'integer': 5, 'z': 5}
```

La línea 1: define una clase muy simple...

Las líneas 3 a la 10: ... la llenan con algunos atributos.

La línea 12: ¡esta es nuestra función!

La línea 13: escanea el atributo __ dict__, buscando todos los nombres de atributos.

La línea 14: si un nombre comienza con i...

La línea 15: ... utiliza la función getattr() para obtener su valor actual; nota: getattr() toma dos argumentos: un objeto y su nombre de propiedad (como una cadena) y devuelve el valor del atributo actual.

La línea 16: comprueba si el valor es de tipo entero, emplea la función isinstance() para este propósito (discutiremos esto más adelante).

La línea 17: si la comprobación sale bien, incrementa el valor de la propiedad haciendo uso de la función setattr(); la función toma tres argumentos: un objeto, el nombre de la propiedad (como una cadena) y el nuevo valor de la propiedad.

Herencia

In [13]:
```
class Star:
    def __init__(self, name, galaxy):
        self.name = name
        self.galaxy = galaxy


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

```
<__main__.Star object at 0x7fae985e8af0>
```

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.

In [14]:
```
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)
```

```
Sol en Vía Láctea
```

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.

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.

Herencia

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.

In [1]:
```
class Vehicle:
    pass


class LandVehicle(Vehicle):
    pass


class TrackedVehicle(LandVehicle):
    pass
```

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 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.

issubclass(): La función devuelve True si ClassOne es una subclase de ClassTwo, y False de lo contrario.

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:

In [2]:
```
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()
```

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

isinstance(): La función devuelve True si el objeto es una instancia de la clase, o False de lo contrario.

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.

In [3]:
```
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()
```

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

is: 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. 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.

In [2]:
```
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(id(string_1))
print(id(string_2))

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

```
False
False
True
1 2 1
140579111872912
140579188182464
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.

In [5]:
```
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)
```

```
Mi nombre es Andy.
```

- 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 llamadaSub. 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.

In [6]:
```
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)
```

```
Mi nombre es Andy.
```

La función super() accede a la superclase sin necesidad de conocer su nombre.

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.

In [7]:
```
class Super:
    supVar = 1


class Sub(Super):
    subVar = 2


obj = Sub()
```

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

```
2
1
```

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.

In [11]:
```
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)
```

```
12
11
```

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.

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).

Herencia Multiple

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

In [12]:
```
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())
```

```
10 11
20 21
```

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.

In [13]:
```
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())
```

```
200 201
```

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.

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.

In [14]:
```
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())
```

```
L LL RR Left
```

La clase Sub hereda todos los bienes de dos superclases Left y Right (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?

Esto prueba que ambos casos poco claros tienen una solución dentro de la clase Left.

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.

In [15]:
```
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(Right, Left): # modificamos las posisciones de Right y left
    pass


obj = Sub()

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

```
R LL RR Right
```

Se observa que los valores de var y fun() provienen de la primera clase.

Jerarquías de clases

In [16]:
```
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()
```

```
do_it de One
do_it de Two
```

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.

Echa un vistazo al código en el editor. 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 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.

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.

In [17]:
```
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)
```

- 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.

Hay un error los métodos turn()son muy similares como para dejarlos en esta forma...

In [18]:
```
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)
```

- 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.

In [19]:
```
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)
```

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

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.

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.

Orden de Resolución de Métodos

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

In [21]:
```
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()
```

```
bottom
middle
top
```

In [22]:
```
class Top:
    def m_top(self):
        print("top")


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


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


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

```
bottom
middle
top
```

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

In [23]:
```
class Top:
    def m_top(self):
        print("top")


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


class Bottom(Top, Middle): # Modificación del orden!!!
    def m_bottom(self):
        print("bottom")


object = Bottom()
```

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[23], line 11
      7     def m_middle(self):
      8         print("middle")
---> 11 class Bottom(Top, Middle):
     12     def m_bottom(self):
     13         print("bottom")

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

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.

Creado por:

Isabel Maniega