Programación Orientada a Objetos - Parte III

1. Polimorfismo: Sobrecarga de métodos (Overriding Methods)

La sobrecarga de métodos se refiere a la posibilidad de que una subclase cuente con métodos con el mismo nombre que los de una clase superior pero que definan comportamientos diferentes.

En el siguiente ejemplo se redefinen dos métodos que son heredados de clases superiores: `__init__` y `reproducirmp3()`

Ejercicio de Inducción: *Pruebe las siguientes líneas de código y observe los resultados:*

```
class Movil(Telefono, Camara, Reproductor):
    def __init__(self):
        print('Móvil encendido')
    def reproducirmp3(self):
        print('Reproduciendo lista mp3')
    def __del__(self):
        print('Móvil apagado')

movil3 = Movil()  # Móvil encendido
movil3.reproducirmp3()  # Reproduciendo lista mp3
del movil3  # Móvil apagado
```

La siguiente lista enumera algunos métodos especiales que se pueden sobreescribir para que tengan comportamientos diferentes:

`__init__ (self [,args...])` Método constructor que se ejecuta al crear un objeto. Sus argumentos son opcionales: objeto = NombreClase(argumentos)

`__del__(self)` Método destructor que se ejecuta al suprimir un objeto: del objeto

`__repr__(self)` Método que se ejecuta cuando se utiliza la función repr(objeto) para convertir datos del objeto a cadenas expresadas como representaciones legibles por el intérprete Python: repr(objeto)

`__str__(self)` Método que se ejecuta cuando imprimimos una instancia del objeto con print(objeto) o llamamos a la función str(objeto) para convertir datos a cadenas imprimibles. A diferencia de `__repr__` la cadena resultante no necesita ser una expresión Python válida: `str(objeto)`

2. Polimorfismo: Sobrecarga de Operadores (Overloading Operators)

La sobrecarga de operadores trata básicamente de lo mismo que la sobrecarga de métodos pero pertenece en esencia al ámbito de los operadores aritméticos, binarios, de comparación y lógicos.

Ejercicio de Inducción: *Pruebe las siguientes líneas de código y observe los resultados:*

```
class Punto:
    def __init__(self,x = 0,y = 0):
        self.x = x
        self.y = y

    def __add__(self,other):
        x = self.x + other.x
        y = self.y + other.y
```

```
        return x, y

punto1 = Punto(4,6)
punto2 = Punto(1,-2)
print(punto1 + punto2)  # (5, 4)
```

3. Ocultamiento de datos (Encapsulamiento)

Los atributos de un objeto pueden ocultarse (superficialmente) para que no sean accedidos desde fuera de la definición de una clase. Para ello, es necesario nombrar los atributos con un prefijo de doble subrayado: **`__atributo`**

Ejercicio de Inducción: *Pruebe las siguientes líneas de código y observe los resultados:*

```
class Factura:
    __tasa = 19

    def __init__(self, unidad, precio):
        self.unidad = unidad
        self.precio = precio

    def a_pagar(self):
        total = self.unidad * self.precio
        impuesto = total * Factura.__tasa / 100
        return(total + impuesto)

compra1 = Factura(12, 110)
print(compra1.unidad)
print(compra1.precio)
print(compra1.a_pagar(), "euros")
print(Factura.__tasa)  # Error:

# AttributeError: type object 'Factura' has no attribute '__tasa'
```

Python protege estos atributos cambiando su nombre internamente. A sus nombres agrega el nombre de la clase: **`objeto._NombreClase__NombreAtributo.`**

Ejercicio de Inducción: *Pruebe las siguientes líneas de código y verifique los resultados presentados:*2

```
print(compra1._Factura__tasa)

# 19
```

4. Propiedades (properties)

Cuando se trabajan con clases es recomendable crear atributos ocultos y utilizar métodos específicos para acceder a los mismos para establecer, obtener o borrar la información:

Ejercicio de Inducción: *Pruebe las siguientes líneas de código y verifique los resultados presentados:*2

```
class Empleado:
    def __init__(self, nombre, salario):
```

```
        self.__nombre = nombre
        self.__salario = salario

    def getnombre(self):
        return self.__nombre

    def getsalario(self):
        return self.__salario

    def setnombre(self, nombre):
        self.__nombre = nombre

    def setsalario(self, salario):
        self.__salario = salario

    def delnombre(self):
        del self.__nombre

    def delsalario(self):
        del self.__salario

empleado1 = Empleado("Francisco", 30000)
print(empleado1.getnombre())
empleado1.setnombre("Francisco José")
print(empleado1.getnombre(), ",", empleado1.getsalario())
```

Estos métodos son útiles principalmente para los atributos más importantes de un objeto, generalmente aquellos que necesitan ser accedidos desde otros objetos.

Pero hay otra alternativa al uso de estos métodos basada en las propiedades Python que simplifica la tarea. Las propiedades en Python son un tipo especial de atributo a los que se accede a través de llamadas a métodos. Con ello, es posible ocultar los métodos **`get`**, **`set`** y **`del`** de manera que sólo es posible acceder mediante estas propiedades por ser públicas.

No es obligatorio definir métodos **`get`**, **`set`** y **`del`** para todas las propiedades. Es recomendable sólo para aquellos atributos en los que sea necesario algún tipo de validación anterior a establecer, obtener o borrar un valor. Para que una propiedad sea sólo de lectura hay que omitir las llamadas a los métodos **`set`** y **`del`**

Ejercicio de Inducción: *Pruebe las siguientes líneas de código y observe los resultados:*

```
class Empleado():
    def __init__(self, nombre, salario):
        self.__nombre = nombre
        self.__salario = salario

    def __getnombre(self):
        return self.__nombre

    def __getsalario(self):
        return self.__salario

    def __setnombre(self, nombre):
        self.__nombre = nombre

    def __setsalario(self, salario):
        self.__salario = salario

    def __delnombre(self):
        del self.__nombre
```

```
    def __delsalario(self):
        del self.__salario

    nombre = property(fget = __getnombre,
                      fset = __setnombre,
                      fdel = __delnombre,
                      doc = "Soy la propiedad 'nombre'")
    salario = property(fget = __getsalario,
                       doc = "Soy la propiedad 'salario'")

empleado1 = Empleado("Francisco José", 30000)
empleado1.nombre = "Rosa"  # Realiza una llamada al método "fset"
print(empleado1.nombre,
      empleado1.salario)  # Realiza una llamada al método "fget"
```

La siguiente asignación no se puede realizar porque la propiedad se ha definido como de sólo lectura. Produce una excepción del tipo **`AttributeError: can't set attribute`**

```
empleado1.salario = 33000
```

Para mostrar la documentación del objeto:

```
help(empleado1)


Help on Empleado in module __main__ object:

class Empleado(builtins.object)
 Methods defined here:
|
 __init__(self, nombre, salario)
|
 ----------------------------------------------------------------------
 Data descriptors defined here:
|
 __dict__
 dictionary for instance variables (if defined)
|
 __weakref__
 list of weak references to the object (if defined)
|
 nombre
 Soy la propiedad 'nombre'
|
 salario
 Soy la propiedad 'salario'
```

---

5. Orden de Resolución de Métodos (MRO). El atributo especial `__mro__`

Es importante conocer cómo funciona la herencia en Python cuando existe una jerarquía con varios niveles de clases que pueden tener definidos métodos que utilizan el mismo nombre.

En el siguiente ejemplo se define un primer nivel de clases con una clase llamada `Clase_A`. A continuación, en un segundo nivel se definen dos clases más (`Clase_A1` y `Clase_A2`) que heredan de la primera. Y en el tercer y último nivel, se define la clase `Clase_X` que hereda de las dos clases de segundo nivel.

Teniendo en cuenta que en las clases mencionadas hay métodos con el mismo nombre, vamos a mostrar cómo calcula Python el orden de resolución de métodos.

Ejercicio de Inducción: *Pruebe las siguientes líneas de código y observe los resultados:*

```
class Clase_A(object):
    def metodo1(self):
        print("Clase_A.metodo1()")

    def metodo3(self):
        print("Clase_A.metodo3()")

    def metodo4(self):
        print("Clase_A.metodo4()")

class Clase_A1(Clase_A):
    def metodo1(self):
        print("Clase_A1.metodo1()")

    def metodo2(self):
        print("Clase_A1.metodo2()")

class Clase_A2(Clase_A):
    def metodo1(self):
        print("Clase_A2.metodo1()")

    def metodo3(self):
        print("Clase_A2.metodo3()")

class Clase_X(Clase_A1, Clase_A2):
    def metodo1(self):
        print("Clase_X.metodo1()")

objeto1 = Clase_X()  # Creación de una instancia (objeto) de Clase_X
objeto1.metodo1()  # Clase_X.metodo1()
objeto1.metodo2()  # Clase_A1.metodo2()
objeto1.metodo3()  # Clase_A2.metodo3()
objeto1.metodo4()  # Clase_A.metodo4()
```

En el ejemplo se crea el objeto objeto1 de la `Clase_X` y después se llama al método `objeto1.metodo1()`. Como dicho método existe en la propia `Clase_X` ese será al que se llame, con independencia de que exista en otra clase.

Como puede comprobarse el método `metodo1()` existe en todas las clases. Si no existiera en la `Clase_X` se hubiera llamado al de la clase `Clase_A1` que tiene mayor prioridad, primero, porque se encuentra en el nivel inmediatamente anterior y, segundo, porque esa clase es nombrada antes que `Clase_A2` en la definición de `Clase_X`.

A continuación, se invoca al método `objeto1.metodo2()` que no existe en la clase `Clase_X` pero si existe

en la clase **Clase_A1**. Como dicho método no existe en otro lugar, ese será el llamado.

Después, se invoca al método **objeto1.metodo3()** que existe tanto en **Clase_A2** como en **Clase_A**. Como la **Clase_A2** se encuentra en el nivel inmediatamente superior (con respecto a la **Clase_X**) ese será el llamado.

Por último, se llama al método **objeto1.metodo4()** que no existe en ninguna clase del nivel inmediatamente superior. Como dicho método está presente en la clase **Clase_A**, ese será el invocado.

La situación puede complicarse si el número de clases aumenta, si hay más niveles y además las clases tienen ancestros diferentes.

En definitiva, dentro de una jerarquía de clases la sobrecarga se resuelve de abajo a arriba y de izquierda a derecha. Si una clase hereda de varias clases se considerará también el orden en que fueron declaradas en la propia definición, es decir, no es igual definir la clase así **[class Clase_X(Clase_A1, Clase_A2)]** que de esta forma: **[class Clase_X(Clase_A2, Clase_A1)]**.

Para calcular el orden de resolución Python utiliza el método MRO (Method Resolution Order) basado en un algoritmo llamado C3.

El cálculo realizado se puede consultar accediendo al atributo especial **__mro__**, que devuelve una tupla con las clases por su orden de resolución de métodos (MRO).

```
print(Clase_X.__mro__)

# (class '__main__.Clase_X', class '__main__.Clase_A1',
#  class '__main__.Clase_A2', class '__main__.Clase_A', class 'object')
```

Si cambiamos en la definición de la clase **Clase_X** el orden de las clases **[class Clase_X(Clase_A2, Clase_A1)]** el resultado de mostrar el atributo **__mro__** será:

```
print(Clase_X.__mro__)
# (class '__main__.Clase_X', class '__main__.Clase_A2',
#  class '__main__.Clase_A1', class '__main__.Clase_A', class 'object')
```

6. La función super()

La función **super()** se utiliza para llamar a métodos definidos en alguna de las clases de las que se hereda sin nombrarla/s explícitamente, teniendo en cuenta el orden de resolución de métodos (MRO). No hay problemas cuando se hereda de sólo una clase, pero si la jerarquía de clases es extensa podemos obtener resultados inesperados si no se tiene un amplio conocimiento de todas las clases y de sus vínculos.

En el siguiente ejemplo se definen las clases **Clase_I** y **Clase_II** con dos métodos cada una, siendo uno de ellos el método constructor o método **__init__**.

A continuación, se definen las clases **Clase_III** y **Clase_IV** que heredan sus métodos y atributos de las clases **Clase_I** y **Clase_II**, pero en cada caso se han establecido con un orden distinto en la definición.

Después, para probar el funcionamiento de la función **super()** se instancian dos objetos de la **Clase_III** y **Clase_IV**, se invocan métodos y se acceden a los atributos. En el propio código se analizan los resultados obtenidos.

Ejercicio de Inducción: *Pruebe las siguientes líneas de código y observe los resultados:*

```
class Clase_I(object):
    def __init__(self):
        self.var1 = 1
        print('Clase_I.__init__')

    def metodo1(self):
```

```
        self.var2 = 1
        print('Clase_I.metodo1()')

class Clase_II(object):
    def __init__(self):
        self.var1 = 2
        print('Clase_II.__init__')

    def metodo1(self):
        self.var2 = 2
        print('Clase_II.metodo1()')

class Clase_III(Clase_I, Clase_II):
    def __init__(self):
        self.var1 = 3
        print('Clase_III.__init__', end = ', ')
        super().__init__()

    def metodo1(self):
        print('Clase_III.metodo1()', end = ', ')
        super().metodo1()
        self.var2 = 3

class Clase_IV(Clase_II, Clase_I):
    def __init__(self):
        self.var1 = 4
        print('Clase_IV.__init__', end = ', ')
        super().__init__()

    def metodo1(self):
        print('Clase_IV.metodo1()', end = ', ')
        super().metodo1()
        self.var2 = 4

# Al crear objeto1 y objeto2 en el método __init__ se
# invoca también el método __init__ de su clase superior

objeto1 = Clase_III()  # Clase_III.__init__, Clase_I.__init__
objeto2 = Clase_IV()  # Clase_IV.__init__, Clase_II.__init__

# El atributo especial __mro__ retorna una tupla
# con las clases ordenadas de izquierda a derecha
# que indican la prioridad en la herencia.
# Mientras en la Clase_III tiene mayor prioridad en
# la herencia la Clase_I que la Clase_II; en la Clase_IV
# es al revés

print(Clase_III.__mro__)
# (class '__main__.Clase_III', class '__main__.Clase_I',
#  class '__main__.Clase_II', class 'object')

print(Clase_IV.__mro__)
# (class '__main__.Clase_IV', class '__main__.Clase_II',
#  class '__main__.Clase_I', class 'object')

# Al llamar al metodo1 de objeto1 y objeto2 se invoca
# también el equivalente de su clase superior
```

```
objeto1.metodo1()  # Clase_III.metodo1(), Clase_I.metodo1()
objeto2.metodo1()  # Clase_IV.metodo1(), Clase_II.metodo1()

# Al acceder a la variable var1 de objeto1 y objeto2
# se obtiene el valor que tiene en la clase superior
# porque en el método __init__ de su clase, después de
# la asignación, se invoca con la función super() al
# método __init__ de la clase superior donde se realiza
# una asignación a la misma variable.

print(objeto1.var1)  # 1
print(objeto2.var1)  # 2

# Al acceder a la variable var2 de objeto1 y objeto2
# se obtiene el valor que tiene en su clase
# porque aunque en el método metodo1() de su clase
# se invoca con la función super() a su equivalente de
# la clase superior, la invocación se realiza ANTES
# de la asignación a dicha variable.

print(objeto1.var2) # 3
print(objeto2.var2) # 4
```