Codo a Codo 4.0

FULL STACK PYTHON

html - css3 - bootstrap - javascript
vue.js - sql - python - django

Clase 31

Python – Parte 7

Herencia

La **herencia** es la capacidad que tiene una clase de heredar los atributos y métodos de otra, ya preexistente, algo que nos permite **reutilizar código**.

Partiremos de una clase sin herencia con muchos atributos y la iremos descomponiendo en otras clases más simples que nos permitan trabajar mejor con sus datos.

Ejemplo sin herencia

Una tienda vende tres tipos de productos: adornos, alimentos y libros. Todos los productos de la tienda tienen una serie de atributos comunes, como la referencia, el nombre, el precio... pero algunos son específicos de cada tipo. Si partimos de una clase que contenga todos los atributos, quedaría más o menos así:

```
class Producto:
    def __init__(self, referencia, tipo, nombre,
                 descripcion, precio, productor="",
                 distribuidor="", isbn="", autor=""):
        self.referencia = referencia #Atributo común
        self.tipo = tipo #Atributo común
        self.nombre = nombre #Atributo común
        self.descripcion = descripcion #Atributo común
        self.precio = precio #Atributo común
        self.productor = productor #Atributo de alimento
        self.distribuidor = distribuidor #Atributo de alimento
        self.isbn = isbn #Atributo de libro
        self.autor = autor #Atributo de libro
```

*Obviamente esto es un despropósito, así que veamos cómo aprovecharnos de la **herencia** para mejorar el planteamiento.*

Superclases

Una superclase es una clase superior o clase base/padre. Dijimos que las clases eran moldes para construir objetos, las superclases son “moldes de moldes”.

A partir de la clase superior puedo construir clases hijas, clases derivadas, cada una con características comunes que comparten atributos con la clase padre, pero que también pueden tener atributos propios.

Así pues, la idea de la herencia es identificar una **clase base** (la **superclase**) con los atributos comunes y luego crear las demás clases heredando de ella (las **subclases**) **extendiendo** sus campos específicos. En nuestro caso esa clase sería el **Producto** en sí mismo.

```
class Producto:
    def __init__(self, referencia, nombre,
                 descripcion, precio):
        self.referencia = referencia
        self.nombre = nombre
        self.descripcion = descripcion
        self.precio = precio

    def __str__(self):
        return "Producto: {} - {} - {} -
 {}".format(self.referencia,self.nombre,self.descripcion,self.precio)
```

Herencia.py

Subclases

Para heredar los atributos y métodos de una clase en otra sólo tenemos que pasarla entre paréntesis durante la definición:

```
class Adorno(Producto):
    pass

adorno = Adorno(2034, "Vaso adornado", "Vaso de porcelana", 15)
print(adorno)
```

```
Producto: 2034 - Vaso adornado - Vaso de porcelana – 15
```
terminal

Como se puede apreciar es posible utilizar el comportamiento de una superclase sin definir nada en la subclase.

Respecto a las demás subclases como se añaden algunos atributos, podríamos definirlas de esta forma:

```
class Alimento(Producto):
    productor = ""
    distribuidor = ""

    def __str__(self):
        return "Referencia: {} \nNombre: {} \nDescripción: {} \nPrecio: {}
\nProductor: {} \nDistribuidor: {}".format(self.referencia, self.nombre, sel
f.descripcion, self.precio, self.productor, self.distribuidor)

class Libro(Producto):
    isbn = ""
    autor = ""

    def __str__(self):
        return "Referencia: {} \nNombre: {} \nDescripción: {} \nPrecio: {}
\nISBN: {} \nAutor: {}".format(self.referencia, self.nombre, self.descripcio
n, self.precio, self.isbn, self.autor)
```

Ahora para utilizarlas simplemente tendríamos que establecer los atributos después de crear los objetos:

```
#Programa principal
alimento = Alimento(2035, "Botella de Aceite de Oliva", "250 ML", 5)
alimento.productor = "La Aceitera"
alimento.distribuidor = "Distribuciones SA"
print(alimento)

libro = Libro(2036, "Cocina Mediterránea","Recetas sanas y buenas", 9)
libro.isbn = "0-123456-78-9"
libro.autor = "Doña Juana"
print(libro)
```

terminal

```
Referencia: 2035
Nombre: Botella de Aceite de Oliva
Descripción: 250 ML
Precio: 5
Productor: La Aceitera
Distribuidor: Distribuciones SA
Referencia: 2036
Nombre: Cocina Mediterránea
Descripción: Recetas sanas y buenas
Precio: 9
ISBN: 0-123456-78-9
```

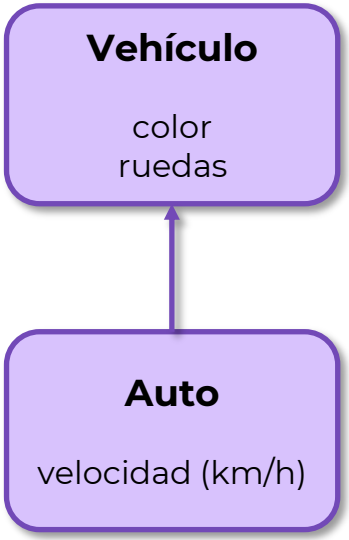
Herencia – Ejemplo: Vehículos y coches

Debemos comenzar por la clase más abstracta.
Vehículo: es una clase superior (superclase) y Coche es una clase concreta (subclase).
Al tener herencia debemos partir de la clase más abstracta:

```
class Vehiculo():

    def __init__(self, color, ruedas):
        self.color= color
        self.ruedas= ruedas
```

Creamos la instancia del objeto (constructor, junto con self y los atributos).



La flecha indica la jerarquía y la relación de herencia.

Recordemos que el ***self*** es para hacer referencia al propio objeto. Cuando lo instancie, cuando llamo al método le indico que estoy hablando de esa única instancia de objeto creada a partir de la clase.

```
def __str__(self):
    return "Color: {} - Ruedas: {}".format(self.color, self.ruedas)
```

Recordemos que **__str__** retorna una cadena, que la vamos formando con las llaves y el format. Esa cadena devuelve información de ese objeto.

Herencia_polimorfismo.py

Para que herede debemos poner **class *claseprincipal(clasequehereda)***. Por ejemplo:

```
class Auto(Vehiculo):
```

Auto *heredará de* ***Vehículo***

Dentro de la clase ***auto*** que deriva de la clase ***vehículo*** tenemos el **__init__** que es el constructor de esa clase heredada, que va a tener los atributos de la superclase (color, ruedas) y los atributos propios (velocidad).

```
def __init__(self, color, ruedas, velocidad):
    Vehiculo.__init__(self, color, ruedas)
    self.velocidad= velocidad
```

Llamada al método constructor de la superclase

Constructor de la clase auto: Se colocan color y ruedas porque derivan de la clase vehículo

La llamada al método constructor de la superclase Vehículo la podemos reemplazar por:

super().__init__(color, ruedas)

```
def __init__(self, color, ruedas, velocidad):
    super().__init__(color, ruedas)
    self.velocidad= velocidad
```

Luego sobreescribimos el método __str__ agregándole la superclase y adjuntándole la velocidad:

```
def __str__(self):
    return super().__str__() + " - Velocidad: " + str(self.velocidad)
```

En el programa principal podemos crear los objetos **Auto** a partir de la superclase **Vehiculo**:

```
# Programa principal
v1= Vehiculo("Blanco",2)
a1= Auto("Rojo",4,140)
a2= Auto("Negro",4,180)
a3= Auto(“Azul",4,200)

print(v1)
print(a1)
print(a2)
print(a3)
```

terminal

```
Color: Blanco - Ruedas: 2
Color: Rojo - Ruedas: 4 - Velocidad: 140
Color: Negro - Ruedas: 4 - Velocidad: 180
Color: Azul - Ruedas: 4 - Velocidad: 200
```

También podríamos agregar los vehículos y autos a una lista de objetos:

```
vehiculos= []
vehiculos.append(v1)
vehiculos.append(a1)
vehiculos.append(a2)
vehiculos.append(a3)

print(vehiculos[0])
print(vehiculos[2])
```

terminal

```
Color: Blanco - Ruedas: 2
Color: Negro - Ruedas: 4 - Velocidad: 180
```

*Los imprime distinto, pero ¿cómo se da cuenta de qué tipo de clase es? Acá empieza a ser importante el **Polimorfismo**.*

Otro ejemplo de herencia:
https://pythones.net/funcion-super-en-python-bien-explicada-ejemplos-oop/

Polimorfismo

El **polimorfismo** es uno de los pilares básicos en la programación orientada a objetos, tiene origen en las palabras ***poly*** (muchos) y ***morfo*** (formas), y aplicado a la programación hace referencia a que los objetos pueden tomar diferentes formas.
Esto significa que los objetos de diferentes clases pueden ser accedidos utilizando el mismo interfaz, mostrando un comportamiento distinto (tomando diferentes formas) según cómo sean accedidos.
En lenguajes de programación como Python, que tiene tipado dinámico, el polimorfismo va muy relacionado con el *duck typing*.
Dado que Python es de tipado dinámico y permite duck typing no es necesario que los objetos compartan un interfaz, simplemente basta con que tengan los métodos que se quieren llamar.

Ejemplo:
Supongamos que tenemos una clase ***Animal*** con un método *hablar()* y por otro lado tenemos otras dos clases, Perro, Gato que heredan de la anterior. Además, implementan el método hablar() de una forma distinta.

```
class Animal:
    def hablar(self):
        pass
```

```
class Gato(Animal):
    def hablar(self):
        print("Miau!")
```

```
class Perro(Animal):
    def hablar(self):
        print("Guau!")
```

A continuación creamos un objeto de cada clase y llamamos al método ***hablar()***. Podemos observar que cada animal se comporta de manera distinta al usar ***hablar()***.

```
for animal in Perro(), Gato():
    animal.hablar()
```

terminal

```
Guau!
Miau!
```

En el caso anterior, la variable animal ha ido "tomando las formas" de Perro y Gato. Sin embargo, nótese que al tener tipado dinámico este ejemplo hubiera funcionado igual sin que existiera herencia entre Perro y Gato.

Fuente del ejemplo: https://ellibrodepython.com/polimorfismo-en-programacion

Polimorfismo

Junto con la herencia, es otra propiedad muy importante en POO. Los métodos son polimorfos. El polimorfismo quiere decir que el método que vaya a implementar / ejecutar a la hora de ser llamado en tiempo de ejecución va a establecer el comportamiento del objeto.
¿Por qué? Porque sabe a qué tipo de objeto está invocándole ese comportamiento, es más dinámico que otros lenguajes donde tengo que ver qué tipo de variable/objeto es como para determinar cómo se muestra. En este caso el propio objeto sabe como "mostrarse" (con su método __str__ que lo imprimo con print).
En este caso no me tengo que detener a ver cómo va a ser implementado ese método, directamente imprimo el objeto y va a saber si lo muestra con la velocidad o sin ella:

```
print(vehiculos[0])
print(vehiculos[2])
```

terminal

```
Color: Blanco - Ruedas: 2
Color: Negro - Ruedas: 4 - Velocidad: 180
```

Muestra información del auto (subclase)

Muestra información del vehículo (superclase)

Puedo iterar sobre la lista vehículos, donde puede observarse el polimorfismo, respondiendo a la implementación del método más cercano (en este caso __str___ que sobreescribe al método __str__ de Vehiculo.

```
# Muestro todos los Vehículos (Vehículo o Auto)
print("Listado de vehículos: ")
for Vehiculo in vehiculos:
    print(Vehiculo)
```

terminal

```
Listado de vehículos:
Color: Blanco - Ruedas: 2
Color: Rojo - Ruedas: 4 - Velocidad: 140
Color: Negro - Ruedas: 4 - Velocidad: 180
Color: Azul - Ruedas: 4 - Velocidad: 200
```

Clases abstractas

Un concepto importante en programación orientada a objetos es el de las ***clases abstractas***. Son clases en las que se pueden definir tanto métodos como propiedades, pero que no pueden ser instanciadas directamente. Solamente se pueden usar ***para construir subclases*** (como si fueran moldes). Permitiendo así tener una única implementación de los métodos compartidos, evitando la duplicación de código.

Propiedades de las clases abstractas

- No pueden ser instanciadas, simplemente proporcionan una interfaz para las subclases derivadas evitando así la duplicación de código.
- No es obligatorio que tengan una implementación de todos los métodos necesarios. Pudiendo ser estos abstractos. Los métodos abstractos son aquellos que solamente tienen una declaración, pero no una implementación detallada de las funcionalidades.
- Las clases derivadas de las clases abstractas deben implementar necesariamente todos los métodos abstractos para poder crear una clase que se ajuste a la interfaz definida. En el caso de que no se defina alguno de los métodos no se podrá crear la clase.

Resumiendo: las clases abstractas definen una interfaz común para las subclases. Proporcionan atributos y métodos comunes para todas las subclases evitando así la necesidad de duplicar código, imponiendo además los métodos que deber ser implementados para evitar inconsistencias entre las subclases.

Clases abstractas

Creación de clases abstractas en Python

Para poder crear clases abstractas en Python es necesario importar la clase ABC y el decorador **abstractmethod** del módulo **abc** (Abstract Base Classes). Un módulo que se encuentra en la **librería estándar** del lenguaje, por lo que no es necesario instalarlo. Así para definir una clase abstracta solamente se tiene que crear una clase heredada de **ABC** con un método abstracto.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def mover(self):
        pass
Animal()
```

abstractmethod *es un paquete con un conjunto de clases / métodos / funciones que podemos reutilizar*

Ahora si se intenta crear una instancia de la clase animal, Python no lo permitirá indicando que no es posible. Es importante notar que, si la clase no hereda de **ABC** o contiene por lo menos un método abstracto, Python permitirá instancias de las clases.

```
TypeError: Can't instantiate abstract class Animal with abstract method mover
```

Clases abstractas

Métodos en las subclases

Las subclases tienen que implementar todos los métodos abstractos, en el caso de que falte alguno de ellos Python no permitirá instanciar tampoco la clase hija:

```
class Animal(ABC):
    @abstractmethod
    def mover(self):
        pass

    @abstractmethod
    def comer(self):
        print("El animal come")
```

Por otro lado, desde los métodos de las subclases podemos llamar a las implementaciones de la clase abstracta con el comando super() seguido del nombre del método. La palabra **pass** permite no definir el contenido de un método.

Ver carpeta clases-abstractas-animales

Clases abstractas

Programa completo

animal.py

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def mover(self):
        pass

    @abstractmethod
    def emitir_sonido(self):
        print("Animal emite sonido:
 ", end="")
```

gato.py

```
from animal import Animal

class Gato(Animal):

    def mover(self):
        print("El gato se mueve.")

    def emitir_sonido(self):
        super().emitir_sonido()
        print("Miau!")
```

perro.py

```
from animal import Animal

class Perro(Animal):

    def mover(self):
        print("El perro se está
 moviendo.")

    def emitir_sonido(self):
        super().emitir_sonido()

        print("Guau, Guau!")
```

main.py

```
from gato import Gato
from perro import Perro

def __main__():
    g1= Gato()
    g1.mover()
    g1.emitir_sonido()
```

sigue...

```
    p1= Perro()
    p1.mover()
    p1.emitir_sonido()

if __name__ == "__main__":
    __main__()
```

Herencia múltiple

La herencia múltiple es la capacidad de una subclase de **heredar de múltiples superclases**.
Esto conlleva un problema, y es que si varias superclases tienen los mismos atributos o métodos, la subclase **sólo podrá heredar de una de ellas**.
En estos casos Python dará prioridad a las clases más a la izquierda en el momento de la declaración de la subclase:

main.py

```
from a import A
from b import B
from c import C

def main():
    c = C()
    c.a()
    c.b()
    c.c()

if __name__ == "__main__":
    main()
```

a.py

```
class A:

    def a(self):
        print("Este método
lo heredo de A")

    def b(self):
        print("Este método
también lo heredo de A")
```

b.py

```
class B:

    def b(self):
        print("Este método
lo heredo de B")
```

c.py

```
from a import A
from b import B

class C(B,A):

    def __init__(self):
        print("Soy de la clase C")

    def c(self):
        print("Este método es de C")
```

Ver carpeta herencia-multiple

Herencia múltiple

Explicación del ejemplo:
main.py es el módulo principal donde creamos la instancia del objeto C. El objeto C heredará los métodos de las clases **A y B**.
La clase **A** tiene dos métodos llamados ***a y b***, mientras que la clase **B** tiene un método llamado ***b***.
El programa principal desde ***main*** crea el objeto C y llama al método ***a*** (de la clase **A**), luego al método ***b***, pero como las clases A y B comparten un método llamado **b** se llama al método de la clase que está más a la izquierda de la subclase C (el método B), por eso en la terminal vemos “Este método lo heredo de B” y no vemos “Este método también lo heredo de A”

main.py

```
from a import A
from b import B
from c import C

def main():
    c = C()
    c.a()
    c.b()
    c.c()

if __name__ == "__main__":
    main()
```

c.py

```
from a import A
from b import B

class C(B,A):

    def __init__(self):
        print("Soy de la clase C")

    def c(self):
        print("Este método es de C")
```

terminal

```
Soy de la clase C
Este método lo heredo de A
Este método lo heredo de B
Este método es de C
```

a.py

```
class A:

    def a(self):
        print("Este método
lo heredo de A")

    def b(self):
        print("Este método
también lo heredo de A")
```

b.py

```
class B:

    def b(self):
        print("Este método
lo heredo de B")
```

Herencia múltiple

Explicación del ejemplo:
SI fuese al revés, es decir si en vez de C(B,A) tuviésemos C(A,B) la terminal nos mostraría “Este método también lo heredo de A” y no “Este método lo heredo de B”.

c.py

```
from a import A
from b import B

class C(B,A):

    def __init__(self):
        print("Soy de la clase C")

    def c(self):
        print("Este método es de C")
```

terminal

```
Soy de la clase C
Este método lo heredo de A
Este método lo heredo de B
Este método es de C
```

≠

c.py

```
from a import A
from b import B

class C(A,B):

    def __init__(self):
        print("Soy de la clase C")

    def c(self):
        print("Este método es de C")
```

terminal

```
Soy de la clase C
Este método lo heredo de A
Este método también lo heredo de A
Este método es de C
```

Diagrama de clases

Es un tipo de diagrama de estructura estática que describe la estructura de un sistema mostrando las clases del sistema, sus atributos, operaciones (o métodos), y las relaciones entre los objetos.
Para especificar la **visibilidad** de un miembro de la clase (es decir, cualquier atributo o método), se coloca uno de los siguientes signos delante de ese miembro:

+	Público
-	Privado
#	Protegido

UML especifica dos tipos de **ámbitos** para los miembros: instancias y clasificadores y estos últimos se representan con nombres subrayados.

- Los miembros **clasificadores** se denotan comúnmente como “estáticos” en muchos lenguajes de programación. Su ámbito es la propia clase.
  - Los valores de los atributos son los mismos en todas las instancias
  - La invocación de métodos no afecta al estado de las instancias
- Los miembros **instancias** tienen como ámbito una instancia específica.
  - Los valores de los atributos pueden variar entre instancias
  - La invocación de métodos puede afectar al estado de las instancias(es decir, cambiar el valor de sus atributos).

Para indicar que un miembro posee un ámbito de clasificador, hay que subrayar su nombre. De lo contrario, se asume por defecto que tendrá ámbito de instancia.

Para ampliar (agregación y composición): https://www.wikiwand.com/es/Diagrama_de_clases

Definición de la función main()

Cuando el intérprete lee un archivo de código, ejecuta todo el código global que se encuentra en él. Esto implica crear objetos para toda función o clase definida y variables globales.

Todo módulo (archivo de código) en Python tiene un atributo especial llamado **__name__** que define el espacio de nombres en el que se está ejecutando. Es usado para identificar de forma única un módulo en el sistema de importaciones.

Por su parte **__main__** es el nombre del ámbito en el que se ejecuta el código de nivel superior (tu programa principal).

El intérprete pasa el valor del atributo **__name__** a la cadena **__main__** si el módulo se está ejecutando como programa principal (cuando lo ejecutas llamando al intérprete en la terminal con python my_modulo.py, haciendo doble clic en él, ejecutándolo en el intérprete interactivo, etc.).

Si el módulo no es llamado como programa principal, sino que es importado desde otro módulo, el atributo **__name__** pasa a contener el nombre del archivo en sí.

Ver carpeta definicion_main y archivo main.py

Ventajas de usar def main()

- El código será más fácil de leer y estará mejor organizado.
- Será posible ejecutar pruebas en el código.
- Podemos importar ese código en un shell de python y probarlo/depurarlo/ejecutarlo.
- Variables dentro def main() son **locales**, mientras que las que están afuera son **globales**. Esto puede introducir algunos errores y comportamientos inesperados.
- Permite ejecutar la función si se importa el archivo como un módulo.

```
from producto import Producto
from alimento import Alimento
from adorno import Adorno
from libro import Libro

def main():
    producto = Producto(2033,"Producto Genérico","1 kg",50)
    alimento = Alimento(2035, "Botella de Aceite de Oliva","250 ML",50,"Marca","Distribuidor")
    adorno = Adorno(2034, "Vaso adornado", "Vaso de porcelana",34,"De Mesa")
    libro = Libro(2036, "Cocina Mediterránea","Recetas buenas",75,"0-123456-78-9","Autor")

if __name__ == "__main__":
    main()
```

Trabajando en conjunto

Gracias a la flexibilidad de Python podemos manejar objetos de distintas clases masivamente de una forma muy simple.

Vamos a empezar creando una lista con nuestros cuatro productos de subclases distintas:

```
productos = [producto,alimento,adorno]
productos.append(libro)
print(productos)
```

terminal

```
[<producto.Producto object at 0x000001F073856BB0>,
 <alimento.Alimento object at 0x000001F073856AC0>,
 <adorno.Adorno object at 0x000001F73856970>,
 <libro.Libro object at 0x000001F073856940>]
```

Ahora si queremos recorrer todos los productos de la lista podemos usar un bucle for:

```
print("Recorriendo lista de productos:")
for producto in productos:
    print(producto)
```

terminal

```
Recorriendo lista de productos:
Producto: 2033 - Producto Genérico - 1 kg - 50
Alimento: 2035 - Botella de Aceite de Oliva - 250 ML - 50
- Marca - Distribuidor
Adorno: 2034 - Vaso adornado - Vaso de porcelana - 34 -
De Mesa
Libro: 2036 - Cocina Mediterránea - Recetas buenas - 75 -
0-123456-78-9 - Autor
```

También podemos acceder a los atributos, siempre que sean compartidos entre todos los objetos:

```
print("Recorriendo lista de productos (atributos):")
for producto in productos:
    print(producto.referencia, producto.nombre)
```

terminal

```
2033 Producto Genérico
2035 Botella de Aceite de Oliva
2034 Vaso adornado
2036 Cocina Mediterránea
```

Si un objeto no tiene el atributo al que queremos acceder nos dará error:

```
for producto in productos:
    print(producto.autor)
```

terminal

```
AttributeError: 'Producto' object
has no attribute 'autor'
```

Por suerte podemos hacer una comprobación con la función *isinstance()* para determinar si una instancia es de una determinada clase y así mostrar unos atributos u otros:

```
print("Condicional isInstance():")
for producto_r in productos:
    if(isinstance(producto_r, Adorno)):
        print(producto_r.referencia, producto_r.nombre, producto_r.estilo)
    elif(isinstance(producto_r, Alimento)):
        print(producto_r.referencia, producto_r.nombre, producto_r.productor)
    elif(isinstance(producto_r, Libro)):
        print(producto_r.referencia, producto_r.nombre, producto_r.isbn)
```

terminal

```
Condicional isInstance():
2035 Botella de Aceite de Oliva Marca
2034 Vaso adornado De Mesa
2036 Cocina Mediterránea 0-123456-78-9
```

Aunque esta no será la forma que utilizaremos a futuro ya que nos valdremos del Poliformismo