ORIENTACIÓN A OBJETOS

Herencia

El concepto de herencia es uno de los mas importantes en la programación orientada a objetos.
En términos generales se trata de establecer una relación entre dos tipos de clases donde las instancias de una de ellas tengan directamente acceso a los atributos y métodos declarados en la otra.

- Para esto debemos contar con una principal que contendrá las declaraciones e implementaciones
- A esta la llamaremos padre, superclase o principal"
- La otra sera la clase hija o secundaria.

La herencia presenta la clara ventaja de la reutilización de código, ademas nos permite establecer relaciones y escribir menos lineas de código, ya que no es necesario, por ejemplo, volver a declarar e implementar métodos.

Python implementa la herencia basándose en los espacios de nombres, de tal forma que, cuando una instancia de una clase hija hace uso de un método o atributo, el interprete busca primero en la definición de la misma y si no encuentra correspondencias accede al espacio de nombres de la clase padre.
Python construye un árbol en memoria que le permite localizar correspondencias entre los diferentes espacios de nombres de clases que hacen uso de la herencia.

Herencia Simple

La herencia simple consiste en que una clase hereda unicamente de otra. Como hemos comentado previamente, la relación de herencia hace posible utilizar, desde la instancia, los atributos de la clase padre.
En Python, al definir una clase, indicaremos entre paréntesis de la clase que hereda.

definimos la clase padre:

```
class Padre ():
    def __init__(self):
            self.x=8
            print("Constructor clase padre")
    def método(self):
            print ("Ejecutando método de clase padre")

>>>
```

Crear una clase que herede de la que acabamos de definir es bien sencillo:

```
 class Hija(Padre):
    def met_hija(self):
          print ("Método clase hija")
>>>
```

Ahora procederemos a crear la instancia de la clase hija y a comprobar como es posible invocar al método definido en su padre:

```
>>> h=Hija()
Constructor clase padre
>>> h.metodo()
Ejecutando metodo de clase padre
>>> h.met_hija()
Método clase hija
>>>
```

importante:
El método **__init__()** de la clase padre ha sido invocado directamente al invocar la instancia de la clase hija.

- Esto se debe a que el método constructor es el primero en ser invocado al crea la instancia y como este existe en la clase padre, entonces se ejecuta directamente.
- Pero ¿que pasa si creamos un método constructor en la clase hija?
- Respuesta: este sera invocado en lugar de llamar al de la clase padre
- Es decir, habremos sobrescrito el constructor original.

Esto puede ser muy útil cuando necesitamos nuestro propio constructor en lugar de llamar al de la clase padre.

Si modificamos nuestra clase Hija, añadimos el siguiente método y volvemos a crear una instancia, podremos apreciar el resultado:

```
>>>
class Hija(Padre):
    def __init__(self):
          print("constructor hija")

>>> z=Hija()
constructor hija
>>>
```

Atributos Heredables
Pero no solo los métodos son heredables, también lo son los atributos. Así la siguiente sentencia es valida:

```
>>>
>>> h.x
8
```

Importante:
Varias clases pueden heredar de otra en común, es decir, una clase padre puede tener varias hijas.

Ej:
Definiremos una clase vehículo que sera la padre y otras dos hijas, llamadas Coche y Moto. La relación que vamos a establecer entre ellas sera la especialización, ya que un coche y una moto son un tipo de vehículo determinado.

clase padre

```
>>>
 class Vehiculo():
    n_ruedas=2
    def __init__(self,marca,modelo):
            self.marca=marca
            self.modelo=modelo
    def acelerar(self):
            pass
    def frenar(self):
            pass
>>>
```

clases hijas

```
>>>
class Moto(Vehiculo):
    pass

class Coche(Vehiculo):
    pass
>>>
```

ahora creamos una instancia de cada clase hija:

```
>>>
>>> c=Coche("porsche","944")
>>> c.n_ruedas=4
>>> m=Moto("Honda","Goldwin")
>>> m.n_ruedas
2
>>>
```

Nota:
1) Como las motos y los coches tienen en común las operaciones de aceleración y frenado, parece lógico que definamos métodos, en la clase padre, para representar dichas operaciones.
2) Por otro lado, cualquier clase que herede de Vehiculo, también contendrá estas operaciones, con independencia del numero de ruedas que tenga.

Herencia Múltiple

Python soporta la herencia múltiple, del mismo modo que C++. Otros lenguajes como Java y Ruby no la soportan, pero si que implementan técnicas para conseguir la misma funcionalidad.
En el caso de Java, contamos con las clases abstractas y las interfaces, y en Ruby tenemos los mixins.

La herencia múltiple es similar en comportamiento a la sencilla con la diferencia que una clase hija tiene uno o mas clases padre.
En Python, basta con separar con comas los nombres de las clases en la definición de las misma.
Por ejemplo, una clase genérica seria Persona, otra Personal y la hija seria Mantenimiento.
De esta forma, una persona que trabajara en una empresa determinada como personal de mantenimiento, podría representarse a través de una clase de la siguiente forma:

```
>>> class Mantenimiento(Personal, Persona):
...     pass
```

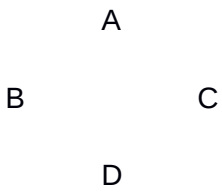
De esta forma, desde la clase Mantenimiento, tendríamos acceso a todos los atributos y métodos declarados, tanto en Persona, como en Personal.

La herencia múltiple presenta el conocido problema del diamante:

- Este problema surge cuando dos clases heredan de otra tercera y, además una cuarta clase tiene como adre a las dos últimas.
- La primera clase padre es llamada A y las clases B y C heredan de ella, a su vez la clase D tiene como padres a B y C.
- En esta situación, si una instancia de la clase D llama a un método definido en A... ***¿lo heredara de la clase B o de la clase C?***
- Cada lenguaje de programación utiliza un algoritmo para tomar esta desición. En el caso de Python, se toma como referencia que todas las clases descienden de la clase padre Object.
- Además, se crea una lista de clases que se buscan de derecha a izquierda y de abajo a arriba, posteriormente se eliminan todas las apariciones de una clase repetida menos la ultima.
- De esta forma queda establecido un orden
- La figura geométrica que representa esta relación entre clases definida anteriormente tiene forma de diamante, de ahí su nombre.

Podemos apreciar esta relación en la figura...

Teniendo en cuenta esta figura, Python crearía la lista de clases D,B,A,C,A
Después eliminaría la primera A, quedando el orden establecido en D,B,C,A



Importante:
1- Dadas las ambigüedades que pueden surgir de la utilización de la herencia múltiple, son muchos los programadores que deciden emplearla lo mínimo posible, ya que, dependiendo de la complejidad del diagrama de herencia, puede ser muy complicado establecer su orden y se pueden producir errores no deseados en tiempo de ejecución.
2- Por otro lado, si mantenemos una relación sencilla, la herencia múltiple es un útil aliado a la hora de representar objetos y situaciones de la vida real.

Polimorfismo

En el ámbito de la orientación a objetos el polimorfismo hace referencia a la habilidad que tiene los objetos de diferentes clases a responder a métodos con el mismo nombre, pero con implementaciones diferentes.
Si nos fijamos en el mundo real, esta situación suele darse con frecuencia. Por ejemplo, pensemos en una serpiente y en un pájaro..
Obviamente ambos pueden desplazarse, pero la manera en la que lo hacen es distinta.
Al modelar esta situación, definiremos dos clases que contienen el método desplazar, siendo su implementación diferente en ambos casos:

```
class Pajaro():
    def desplazar(self):
            print("Volar")

class Serpiente():
    def desplazar(self):
            print("Raptar")
>>>
```

A continuación, definiremos una función adicional que se encargue de recibir como argumento la instancia de una de las dos clases y de invocar al método del mismo nombre. Dado que ambas clases contienen el mismo método, la llamada funcionará sin problema, pero el resultado sera diferente.

```
 def mover(animal):
     animal.desplazar()
...
>>> p=Pajaro()
>>> s=Serpiente()
>>>
>>> p.desplazar()
Volar
>>> s.desplazar()
Raptar
>>> mover(p)
Volar
>>> mover(s)
Raptar
>>>
```

nota:
En Python la utilización de polimorfismo es bien sencilla. Esto se debe a que no es un lenguaje fuertemente tipado.
-Otros lenguajes que si lo son utilizan técnicas para permitir y facilitar el uso del polimorfismo.
-Ej: Java cuenta con las clases abstractas, que permiten definir un método sin implementarlo.
-Otras clases pueden heredar de una clase abstracta e implementar el método en cuestión de forma diferente.

Introspección

La introspección o inspección interna es la habilidad que tiene un componente, como una instancia o un método, para examinar las propiedades de un objeto externo y tomar decisiones sobre las acciones que el mismo puede llevar a cabo, basándose en la información conseguida a través de la examinación.

- Python posee diferentes herramientas para facilitar la introspección, lo que puede resultar muy útil para conocer que acciones es capaz de realizar un objeto determinado.
- Una de las mas populares de estas herramientas es la función integrada dir()
- Que nos devuelve todos los métodos que pueden ser invocados para una instancia concreta.

De esta forma, sin necesidad de invocar ningún método, tenemos información sobre un objeto.

Ej:
definamos una clase con un para de métodos y lancemos la función dir() sobre una instancia concreta:

```
class Intro():

    def __init__(self, val):
            self.x=val
    def primero(self):
            print("Primero")
  def segundo(self):
            print("Segundo")

>>> i=Intro("Valor")
>>> dir(i)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'primero', 'segundo', 'x']
```

nota:
La salida de la sentencia dir(i) es un lista de strings con el nombre de cada uno de los métodos que pueden ser invocados.
Los tres últimos son los métodos de instancia y el atributo que hemos creado, mientras que el resto son métodos heredados de la clase predefinida object.

Otras

Además de **dir()**, otras dos prácticas funciones de introspección son **isinstance()** y **hasttr()**

isinstance(), nos sirve para comprobar si una variable es instancia de una clase. En caso afirmativo devolverá True y en otro caso False.

Ej:

```
class Intro():
    def __init__(self,val):
            self.x=val
    def primero(self):
            print("Primero")
    def segundo(self):
            print("segundo")

>>> i=Intro("mariano")
>>>
>>> isinstance(i, Intro)
True
```

También dirá siempre que es instancia de object ya que todas heredan de object

```
>>> isinstance(i, object)
True
```

si creamos otra clase y preguntamos por la instancia de la clase Intro

```
class otraclase():
    pass
>>> isinstance(i, otraclase)
False
>>>
```

hasattr()

Devuelve True si una instancia contiene un atributo. Esta función recibe como primer parámetro la variable que representa la instancia y como segundo el atributo por el que deseamos preguntar.

Ej:

Dada nuestra clase anterior, escribiremos el siguiente código para comprobar su funcionamiento.

```
>>> if(i,'x'):print("Instancia: i. Clase: Intro. Atributo: x")
...
Instancia: i. Clase: Intro. Atributo: x
>>> if(i,'z'):print("Atributo z no existe")
...
Atributo z no existe
>>>
```

importante:

-Si utilizamos la herencia, las funciones de introspección pueden sernos muy útiles, ademas nos permiten descubrir como los métodos y atributos son heredados.

-Si creamos una nueva clase que herede de la anterior, pero con un nuevo método, al creamos una nueva clase que herede de la anterior, pero con un nuevo método, al llamar a dir() veremos como los atributos y métodos están disponibles directamente:

```
class Hijo(Intro):
    def tercero(self):
        print("Tercero")

>>> h=Hijo("hijo")
>>> dir(h)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'primero', 'segundo', 'tercero', 'x']
>>>
>>>
```

De igual forma, podemos emplear la función hasattr() para comprobar si el atributo x existe también en la clase h:

```
>>> if hasattr(h,'x'):print("h puede acceder a x")
...
h puede acceder a x
>>>
```

Obviamente, aunque las clases Intro e Hijo estén relacionadas son diferentes. Sin embargo, la función isinstance() nos devolverá True si pasamos como primer argumento una instancia de cualquiera de ellas y como segundo el nombre de una de las dos clases.

```
>>> if isinstance(h,Intro): print("h es instancia de Intro")
...
h es instancia de Intro
>>>
```

Entonces: ¿como podemos averiguar si una instancia pertenece a la clase padre o hija?
Respuesta: basta con emplear la función integrada type(), como se muestra:

```
>>>
>>> type(i)
<class '__main__.Intro'>
>>> type(h)
<class '__main__.Hijo'>
>>>
```

Callable()
Otro método relacionado con la introspección es callable(), el cual nos devuelve True si un objeto puede ser llamado y False en caso contrario.
Por defecto, todas las funciones y métodos de objetos pueden ser llamados, pero no las variables.
Gracias a este método podemos averiguar, por ejemplo, si una determinada variable es una función o no.

Ej:

```
>>> cad="cadena"
>>> callable(cad)
False
 def fun():
    return ("fun")

>>> callable (fun)
True
>>>
```

Siguiendo la misma explicación, una instancia de objeto devolverá False cuando invocamos a callable()

```
>>> callable(i)
False
>>>
```

Importante:
Mirando el resultado de la ejecución. de la sentencia previa dir(h), descubriremos métodos que pueden ser llamados a través de una instancia y que también sirven para la introspección.
Nos referimos a **__getattribute__()** y **__setattr__()**

__getattribute__()
nos sirve para obtener el valor de un atributo a través de su nombre
__setattr__()
este realiza la operación inversa, es decir, recibe como parámetro el nombre del atributo y el valor que va a serle fijado.

Nota:
En general estos métodos no son necesarios, ya que podemos acceder directamente a un atributo de instancia directamente usando el nombre de la variable de instancia, seguido de un punto y del nombre del atributo en cuestión.
Sin embargo, los métodos __getattribute__() y __setattr__() pueden ser muy útiles, cuando, por ejemplo, necesitamos leer los atributos de una lista o diccionario.

Ej: Supongamos que tenemos una clase con tres atributos de instancia definidos y deseamos fijar su valor empleando para ello un diccionario.

1) definimos la clase

```
class Test():
    def __init__(self):
            self.x=3
            self.y=4
            self.z=5
>>>
```

2) Ahora crearemos un diccionario y una instancia de clase

```
>>> attrs={"x":1,"y":2,"z":3}
>>>
```

3) Instanciamos un objeto de la clase Test

```
>>> t=Test()
```

4) Ahora pasaremos a asignar los valores del diccionario a nuestros atributos de clase

```
 for k, v in attrs.items():
     t.__setattr__(k,v)

>>>
```

5) Para comprobar que los atributos han sido fijados correctamente, emplearemos la función **__getattribure__()**, asi:

```
for k in attrs.keys():
   print("{0}={1}".format(k,t.__getattribute__(k))

>>>
Y=2
X=1
Z=3
```

asi vemos el diccionario

```
>>> print(attrs)
{'y': 2, 'x': 1, 'z': 3}
>>>
```

asi vemos que los valores de la instancia fueron cambiados

```
>>> print(t.x,t.y,t.z)
1 2 3
>>>
```

Nota:
Hay que tener en cuenta que **attrs** es un diccionario y como tal es una estructura de datos no ordenada, por lo cual la salida del orden de los atributos puede diferir durante sucesivas ejecuciones de la iteración.