ORIENTACIÓN A OBJETOS

Métodos Especiales

Python pone a nuestra disposición una serie de métodos especiales que, por defecto, contendrán todas las clases que creemos.

- En realidad cualquier clase que creemos sera hija de una clase especial integrada llamada object.
- Así, dado que esta clase cuenta por defecto con estos métodos especiales, ambos estarán disponibles en nuestros objetos.
- Podemos sobrescribir estos métodos reemplazando su funcionalidad.

Cuando creamos una nueva clase, esta hereda directamente de object, por lo que no es necesario indicar esto explícitamente al instanciar un objeto de una clase concreta.

Los métodos especiales se caracterizan principalmente porque utilizan dos caracteres underscores al principio y al final del nombre del metodo.

Creación e Inicialización

El principal de estos métodos especiales ya lo hemos visto... se trata de:

__init__()

- Este método será el encargado de realizar las tareas de inicialización cuando la instancia de una clase determinada es creada.
- Necesita utilizar como parámetro formal una referencia (self) a la instancia y, adicionalmente, pueden contener otros parámetros.

__new__()

En muchos lenguajes la operación de creación e inicialización de instancia es atómica.
En Python esto es diferente:

- Primero se invoca a __new__() para crear la instancia propiamente dicha
- Posteriormente se llama a __init__() para llevar a cabo las operaciones que sean requeridas para inicializar la misma.

Nota:
- En la practica suele utilizarse __init__() como constructor de instancia, de la misma forma que en Java, Por ejemplo, se emplea un método que tiene el mismo nombre que la clase que lo contiene.
- Gracias a este mecanismo de Python podemos tener mas control sobre las operaciones de creación e inicialización de objetos.
- Dado que cualquier clase los contendrá al heredar de object, estos metodos podrán ser sobrescritos con la funcionalidad que deseemos.

Importante:
A diferencia de __init__() , __new__() no requiere de self, es su lugar recibe una referencia a la clase que lo esta invocando.
---Realmente, es este un metodo estático que siempre devuelve un objeto---

El metodo __new__() fue diseñado para permitir la personalización en la creación de instancias de clases que heredan de aquellas que son inmutables.
--Recordemos que para Python, algunos tipos integrados son inmutables, como, por ejemplo, los strings y las tuplas--

Es importante tener en cuenta que el metodo __new__() solo llamara automáticamente a __init__() si el primero devuelve una instancia.

Es practica habitual realizar ciertas funciones de personalización dentro de __new__ y luego llamar al metodo del mismo nombre de la clase padre. De esta forma aseguramos que nuestro metodo __init__() sera ejecutado.

Ej:

```
class Ini():
    def __new__(cls):
            print("New")
            return super(Ini,cls).__new__(cls)
    def __init__(self):
            print("init")
>>>
```

Ahora crearemos una instancia y observaremos el resultado producido que nos indicara el orden en el que ambos métodos han sido ejecutados:

```
>>> obj=Ini()
New
init
>>>
```

Nota:
¿Que pasaría si el metodo __new__() de la clase anterior no devolviera una instancia?
Respuesta: nuestro metodo __init__() nunca seria ejecutado.

Ej:
Realizar esta prueba es bastante sencilla, basta con borrar la ultima linea, la que contiene la sentencia return del metodo __new__.
Al volver a crear la instancia comprobaremos como solo obtenemos como salida la cadena de texto "init".

```
class Ini():
    def __new__(cls):
          print("New")
    def __init__(self):
          print("init")

>>> obj=Ini()
New
>>>
```

New Con Parámetros Adicionales

Cuando trabajamos con __new__() y este recibe parámetros adicionales, estos mismos deben constar como parámetros formales en el metodo __init_()

Ej:

```
class Ini2():
    def __new__(cls,x):
          return super(Ini2,cls).__new__(cls)
    def __init__(self,x):
          self.x=x


>>> T=Ini2("hola")
>>> T.x
'hola'
>>>
```

Destructor

Al igual que otros lenguajes de programación, Python cuenta con un metodo especial que puede ejecutar acciones cuando el objeto va a ser destruido.
Python incorpora un recolector de basura (garbage collector) que se encarga de llamar a este destructor y liberar memoria cuando el objeto no es necesario.
--Este proceso es transparente y no es necesario invocar al destructor para liberar memoria--
Sin embargo, este metodo destructor puede realizar acciones adicionales si lo sobrescribimos en nuestras clases.
Su nombre es **__del__()** y habitualmente nunca se llama explícitamente.

Importante:
1- La función integrada del() no llama directamente al destructor de un objeto, si no que esta sirve para decrementar el contador de referencias al objeto.
2- El metodo especial __del__() es invocado cuando el contador de referencias llega a cero.

Ej:
Usaremos la clase anterior Ini(), le añadiremos el siguiente metodo y luego crear una nueva instancia y observar como el destructor es llamado automáticamente por el interprete.
El código a añadir es:

```
def __del__(self):
                print("del..")
```

pasos:
1- En lugar de utilizar directamente el interprete a través de la consola de comandos, vamos a crear un fichero que contenga nuestra clase completa, incluyendo el destructor.
2- Seguidamente invocaremos al interprete e Python para que ejecute nuestro script y comprobaremos como la salida nos muestra la cadena "del.." como consecuencia de la ejecución del destructor por parte del recolector de basura.
3- Al terminar la ejecución del script y no ser necesario el objeto, el interprete invoca al recolector de basura, que a su vez, llama directamente al constructor.

Representación Y Formatos

En algunas ocasiones puede ser útil obtener una representación de un objeto, que nos sirva, por ejemplo, para volver a crear un objeto con los mismos valores que el original.
La representación de un objeto puede obtenerse a través de la función integrada **repr()**.

Ej: si declaramos un string con un valor determinado e invocamos a la mencionada función, conseguiremos el valor de la cadena.
En el caso de un string su valor es lo único que necesitamos para recrear el objeto.

```
>>> a="mariano"
>>> repr(a)
'"mariano"'
>>>
```

La situación cambia cuando creamos nuestras propias clases. En este caso, deberemos emplear el método especial **__repr__()**, el cual sera invocado directamente cuando se llama a la función repr()

Ej:
Pensemos en una clase que representa un coche, donde sus atributos principales son la marca y el modelo. Dado que estos son los únicos valores que necesitamos para recrear el objeto, serán los que utilizaremos en nuestro metodo **__repr__()**
Así:

```
class Coche:
    def __init__(self, marca="Audi", modelo="A4"):
            self.marca=marca
            self.modelo=modelo
    def __repr__(self):
            return("{0}-{1}".format(self.marca, self.modelo))
>>>
```

Al crear un objeto e invocar a la mencionada función de representación obtendremos el siguiente resultado:

```
>>c=Coche()
>>> print(repr(c))
Audi-A4
>>>
```

Crear Un Nuevo Objeto

Con la información devuelta por la función **repr()** podríamos crear un nuevo objeto con los mismos valores que el original:

```
>>> arr=repr(c).split("-")
>>> d=Coche(arr[0],arr[1])
>>> print(repr(d))
Audi-A4
>>>
```

El metodo especial __repr__() siempre debe devolver un string, en caso contrario se lanzara un excepción de tipo TypeError.
--Habitualmente, esta representación de un objeto se emplea para poder depurar código y encontrar posibles errores--

Relacionando Los Métodos __repr__() con __str__()

Relacionado con __repr__() encontramos otro metodo especial denominado __str__(), el cual es invocado cuando se llama a las funciones integradas str() y print(), pasando como argumento un objeto.

1) El metodo __str__() puede ser considerado también una representación del objeto en cuestión, la diferencia con __repr__() radica en que el primero debe devolver una cadena de texto que nos sirva para identificar y representar al objeto de una formas sencilla y concisa.

--La información que devuelve suele ser una linea de texto descriptiva--

2) Al igual que __repr__(), el metodo __str__() debe devolver siempre un string.

Ej:
con la clase anterior

```
>>> c=Coche()
>>> print(repr(c))
Audi-A4
class Coche:
    def __init__(self, marca="Audi", modelo="A4"):
            self.marca=marca
            self.modelo=modelo
    def __repr__(self):
            return ("{0}-{1}".format(self.marca, self.modelo))
```

A la clase le agregamos..

```
 def __str__(self):
            return ("{0}->{1}".format(self.marca, self.modelo))
>>>
>>> d=Coche()
```

cuando llamemos a la función print nos devolverá:

```
>>> print(d)
Audi->A4
>>>
```

Importante:
En nuestro ejemplo para la clase Coche no existe mucha diferencia entre las cadenas que devuelven ambos métodos de representación; sin embargo, si la clase es muy compleja, si que es mas fácil establecer diferencias entre los resultados devueltos.

Metodo Especial __bytes__()

Similar a __str__() también existe el metodo especial __bytes__() que devuelve también una representación del objeto utilizando el tipo predefinido bytes.
Este metodo sera ejecutado cuando llamamos a la función integrada bytes(), pasando como argumento un objeto.

Metodo Especial __format__()

Es otro metodo especial utilizado para obtener una representación de una formato determinado de un objeto.
Es decir, podemos indicar, valores como, por ejemplo, la alineación de la cadena de texto producida.
- Esto nos ayudara a crear una cadena de texto convenientemente formateada.
- La función integrada format() es la responsable de llamar a este metodo especial, y al igual que los otros métodos de representación, requiera pasar la instancia de la clase en cuestión.

```
>>> format(d)
'Audi->A4'
>>>
```

Comparaciones

Comparar tipos sencillos es fácil. Por ejemplo, si tenemos dos números, determinar si uno es mayor que otro es trivial. Igual ocurre para otras operaciones similares como son la igualdad, la comprobación de si es igual o mayor que y la desigualdad.

Ahora, esto cambia cuando debemos aplicar estas operaciones a objetos de nuestras clases.
Ello se debe a que el interprete, a priori, no sabe como realizar estas operaciones.
Para solventar este problema, Python cuenta con una serie de métodos especiales que nos ayudaran a escribir nuestra propia lógica aplicable a cada operación de comparación concreta.

--Bastara con implementar el metodo que necesitemos sobrescribiendo el original ofrecido por el lenguaje para esta situación.--

Los métodos especiales de comparación son:
__lt__() : Menor que
__le__() : Menor o igual que
__gt__() : Mayor que
__ge__() : Mayor o igual que
__eq__() : Igual a
__ne__() : Distinto de

Cada uno de estos métodos sera invocado en función del operador equivalente que empleemos en la comparación.

Ej:
Supongamos que deseamos saber que modelo de coche es mas moderno. Por simplicidad, nos basaremos en el atributo que representa la marca y tendremos en cuenta que ese solo puede ser un numero.
-Cuanto mayor es el numero, mas moderno sera el coche en cuestión.
En base a esta simple afirmación, escribiremos el código del metodo que se encargara de realizar la comprobación:

```
>>> class Coche:
...     def __init__(self, marca="Audi", modelo="A4"):
...             self.marca=marca
...             self.modelo=modelo
...     def __repr__(self):
...             return ("{0}->{1}".format(self.marca, self.modelo))
```

añadimos el metodo a nuestra clase Coche

```
...     def __gt__(self,objeto):
...             if int (self.modelo)> int(objeto.modelo):
...                     return True
...             return False
...
>>>
```

Creamos 2 Instancias, y podremos emplear el operador >> para llevar a cabo nuestra prueba:

```
>>> modelo_A4=Coche("Audi",A4)
>>> modelo_A5=Coche("Audi",A5)
>>>
```

La prueba en la consola seria:

```
>>>
>>> if modelo_A5 > modelo_A4:
...     print ("El {0} es un modelo superior".format(modelo_A5.modelo))
...
El A5 es un modelo superior
>>>
```

De forma análoga podemos emplear el resto de métodos especiales de comparación.
Es importante tener en mente que estos métodos deben devolver True o False.
--Sin embardo, esto es solo una convención, ya que, en realidad, pueden devolver cualquier valor--

Hash Y Bool

__hash__()
Se ejecuta al invocar a la función integrada hash() y debe devolver un numero entero que sirve para identificar de forma univoca a cada instancia de la misma clase.

- De esta forma es fácil comparar si dos objetos son el mismo, ya que deben tener el mismo valor devuelto por hash().
- A través del metodo especial __hash__(), podemos realizar operaciones que nos sean necesarias y devolver después el correspondiente valor.
-

Por defecto todos los objetos de cualquier clase cuentan con los métodos __eq__() y __hash__() y siempre dos instancias serán diferentes a no ser que se comparen consigo mismas.

Ej: ejemplo de invocación al metodo __hash__() desde dos instancias diferentes.

primer objeto..

```
 class TestHash():
    pass

>>> t=TestHash()
>>> t.__hash__()
-9223363288826460903
>>> hash(t)
-9223363288826460903
>>>
```

Segundo objeto...

```
>>> x=TestHash()
>>> x.__hash__()
-9223363288826470021
>>> hash(x)
-9223363288826470021
>>>
```

Podemos comparar los objetos...

```
>>>
>>> x==x
True
>>> t==t
True
>>> t==x
False
>>>
```

Metodo Especial __bool__()

La función bool() sera la encargada de llamar al metodo especial __bool__() cuando esta recibe como argumento la instancia de una clase determinada.

- Por defecto, si no implementamos el mencionado metodo en nuestra clase, la llamada a bool() siempre devolverá True.
- Si implementamos __bool__() en nuestra clase, este debe devolver un tipo booleano, es decir, en la practica, solo podremos devolver True o False.

Ej:
sobrescribiremos el metodo __bool__() para que devuelva False, cambiando así el valor que tiene por defecto cuando no esta implementado:

```
>>> class TestBool():
...     def __bool__(self):
...             return False
...
>>> t=TestBool()
>>> t.__bool__()
False
>>>
```

importante:
---si no se le pasa self a __bool__() da error de argumentos---

```
if t: print("Es Falso")
 else: print("no es falso..")

no es falso..
>>>
```