Python

Guía para ser un Pythonista

Juan José Lozano Gómez – j2logo.com

15

Programación orientada a objetos en Python

Índice de contenidos

Introducción

De todos los temas que hemos visto hasta ahora, diría que este es uno de los más importantes: *Programación Orientada a Objetos en Python*. Y es que, como te he mencionado en varias ocasiones, en Python todo es un objeto. Si dominas los conceptos que describo en este artículo, estarás un paso más cerca de ser un auténtico Pythonista.

Como sabrás, Python es un lenguaje multiparadigma: soporta la programación imperativa y funcional, pero también la programación orientada a objetos.

La verdad es que el tema da para mucho. Por eso, aquí, repasaremos en detalle los conceptos clave de la programación orientada a objetos desde el punto de vista de Python.

Python es un lenguaje orientado a objetos

Sí, soy un pesado y por eso te lo vuelvo a decir: **En Python todo es un objeto**. Cuando asignas un valor entero a una variable, ese valor es un objeto; una función es un objeto; las listas, tuplas, diccionarios, conjuntos, ... son objetos; una cadena de caracteres es un objeto. Y así podría seguir indefinidamente.

Pero ¿por qué es tan importante la programación orientada a objetos? Bien, este tipo de programación introduce un nuevo paradigma que permite encapsular en una misma entidad datos y operaciones que se pueden realizar sobre dichos datos. **Su máxima es la reutilización del código**.

En la programación imperativa (o estructurada), uno de los elementos principales son las funciones. Además, el flujo de un programa se ejecuta de manera secuencial. Sin embargo, en la programación orientada objetos el elemento principal son los objetos y, aunque el flujo del programa sigue siendo secuencial, en realidad este está determinado por el comportamiento de los objetos.

Sigue leyendo para que entiendas qué quiero decir.

Clases y objetos

Básicamente, una clase es una entidad que define una serie de elementos que determinan un **estado** (datos) y un **comportamiento** que modifica dicho estado (operaciones sobre los datos).

Por su parte, **un objeto es una concreción o instancia de una clase**.

Tranqui, que lo vas a entender con el siguiente ejemplo.

Seguro que, si te digo que te imagines un coche, en tu mente comienzas a visualizar la carrocería, el color, las ruedas, el volante, si es diésel o gasolina, el color de la tapicería, si es manual o automático, si acelera o va marcha atrás, etc.

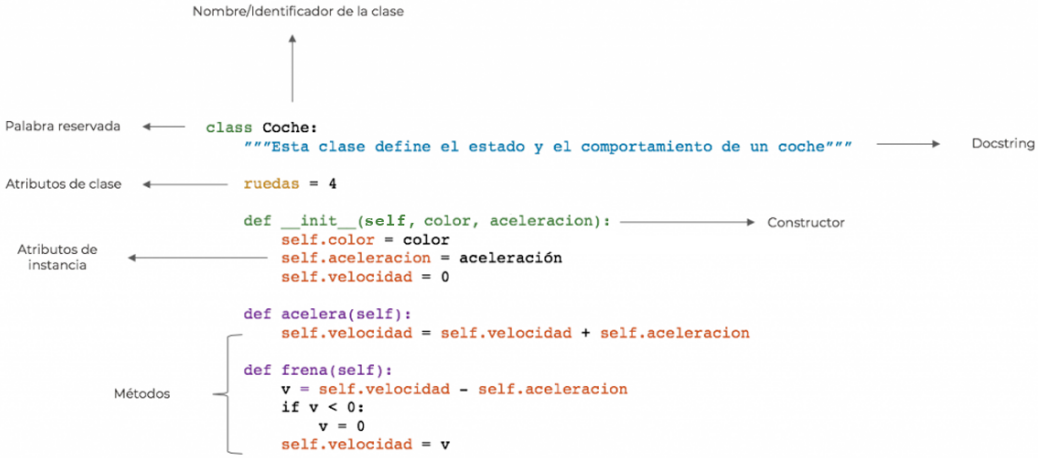
Pues todo lo que acabo de describir viene a ser una clase y cada uno de los de coches que has imaginado, serían objetos de dicha clase.

¿Cómo pasamos lo anterior a Python? Veámoslo.

Como te decía, una clase engloba datos y funcionalidad (o comportamiento). Cada vez que se define una clase en Python, se crea a su vez un tipo nuevo (¿recuerdas?

tipo `int`, `float`, `str`, `list`, `tuple`, ... todos ellos están definidos en una clase).

Para definir una clase en Python se utiliza la palabra reservada `class`. El siguiente esquema visualiza los elementos principales que componen una clase. Todos ellos los iremos viendo con detenimiento en las siguientes secciones:



El esquema anterior define la clase `Coche` (es una versión muy, muy simplificada de lo que es un coche, pero nos sirve de ejemplo). Dicha clase define una serie datos, como `ruedas`, `color`, `aceleración` o `velocidad` y las operaciones `acelera()` y `frena()`.

IMPORTANTE: Al definir una clase, se crea un nuevo espacio de nombres y se usa como ámbito local por dicha clase; por tanto, todas las definiciones y asignaciones a variables dentro de la misma, se asocian a este nuevo espacio de nombres.

Además, al ámbito local original (suele ser el ámbito global de un módulo) se añade el nombre de la clase, el cuál está asociado a un objeto de tipo clase basado en la definición de la clase que se acaba de crear.

Sobre una clase solo se pueden aplicar dos tipos de operaciones: **referenciar a un atributo** e **instanciar**.

Las referencias de atributos utilizan la sintaxis estándar de Python: `obj.atributo`.

En el ejemplo de la clase `Coche`, se podría referenciar a cualquiera de los elementos definidos dentro de la clase, por ejemplo:

```
>>> Coche.ruedas
4
```

Cuando se instancia una clase lo que se hace es crear un objeto de dicha clase, una instancia. Como ya hemos visto, normalmente, el objeto se asigna a una variable que hace referencia al objeto creado.

En el siguiente ejemplo se crean dos objetos de tipo `Coche`:

```
>>> c1 = Coche('rojo', 20)
>>> print(c1.color)
rojo
>>> print(c1.ruedas)
4
>>> c2 = Coche('azul', 30)
>>> print(c2.color)
azul
>>> print(c2.ruedas)
4
```

`c1` y `c2` son dos variables que referencian a objetos, objetos cuya clase es `Coche`. Ambos objetos pueden *acelerar* y *frenar*, porque su clase define estas

operaciones y tienen un *color*, porque la clase `Coche` también define este dato. Lo que ocurre es que el objeto referenciado por `c1` es de color rojo, mientras que el referenciado por `c2` es de color azul.

¿Ves ya la diferencia?

NOTA: Es una convención utilizar la notación *CamelCase* para los nombres de las clases. Esto es, la primera letra de cada palabra del nombre está en mayúsculas y el resto de letras se mantienen en minúsculas.

Constructor de una clase

En la sección anterior me he adelantado un poco... Para crear un objeto de una clase determinada, es decir, instanciar una clase, se usa el nombre de la clase y a continuación se añaden paréntesis (como si se llamara a una función).

```
class MiClase:
    pass

obj = MiClase()
```

El código anterior crea una nueva instancia de la clase `MiClase` y asigna dicho objeto a la variable `obj`. Esto crea un objeto vacío, sin estado (la clase no define ningún atributo).

Sin embargo, hay clases (como nuestra clase `Coche`) que deben o necesitan crear instancias de objetos con un estado inicial.

Esto se consigue implementando el método especial `__init__()`. Este método es conocido como el constructor de la clase y se invoca cada vez que se instancia un nuevo objeto.

A su vez, el método `__init__()` establece un primer parámetro especial que se suele llamar `self` (veremos qué significa este nombre en la siguiente sección). Pero puede especificar otros parámetros siguiendo las mismas reglas que cualquier otra función.

En nuestro caso, el constructor de la clase coche es el siguiente:

```
def __init__(self, color, aceleracion):
    self.color = color
    self.aceleracion = aceleracion
    self.velocidad = 0
```

Como puedes observar, además del parámetro `self`, define los parámetros `color` y `aceleracion`, que determinan el estado inicial de un objeto de tipo Coche.

En este caso, para instanciar un objeto de tipo `Coche`, debemos pasar como argumentos el color y la aceleración como vimos en el ejemplo:

```
c1 = Coche('rojo', 20)
```

IMPORTANTE: A diferencia de otros lenguajes, en los que está permitido implementar más de un constructor, en Python solo se puede definir un método `__init__()`.

Atributos, atributos de datos y métodos

Una vez que sabes qué es un objeto, tengo que decirte que la única operación que pueden realizar los objetos es referenciar a sus atributos por medio del operador `.`.

Como habrás podido apreciar, un objeto tiene dos tipos de atributos: *atributos de datos* y *métodos*.

Los atributos de datos definen el estado del objeto. En otros lenguajes son conocidos simplemente como atributos o miembros.

Los métodos son las funciones definidas dentro de la clase y **definen el comportamiento de un objeto**.

Siguiendo con nuestro ejemplo de la clase `Coche`, observa el siguiente código:

```
>>> c1 = Coche('rojo', 20)
>>> print(c1.color)
rojo
>>> print(c1.velocidad)
0
>>> c1.acelera()
>>> print(c1.velocidad)
20
```

En la línea 2 del código anterior, el objeto `c1` está referenciando al atributo de dato `color` y en la línea 4 al atributo `velocidad`. Sin embargo, en la línea 6 se referencia al método `acelera()`. Llamar a este método tiene una implicación como puedes observar y es que modifica el estado del objeto, dado que se incrementa su velocidad. Este hecho lo puedes apreciar cuando se vuelve a referenciar al atributo `velocidad` en la línea 7.

Atributos de datos

A diferencia de otros lenguajes, los atributos de datos no necesitan ser declarados previamente. Un objeto los crea del mismo modo en que se crean las variables en Python, es decir, cuando les asigna un valor por primera vez.

El siguiente código es un ejemplo de ello:

```
>>> c1 = Coche('rojo', 20)
>>> c2 = Coche('azul', 10)
>>> print(c1.color)
rojo
>>> print(c2.color)
azul
>>> c1.marchas = 6
>>> print(c1.marchas)
6
>>> print(c2.marchas)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'Coche' object has no
attribute 'marchas'
```

Los objetos `c1` y `c2` pueden referenciar al atributo `color` porque está definido en la clase Coche. Sin embargo, solo el objeto `c1` puede referenciar al atributo `marchas` a partir de la línea 7, porque inicializa dicho atributo en esa línea. Si el objeto `c2` intenta referenciar al mismo atributo, como no está definido en la clase y tampoco lo ha inicializado, el intérprete lanzará un error.

Métodos

Como te explicaba al comienzo de esta sección, los métodos son las funciones que se definen dentro de una clase y, por consiguiente, pueden ser referenciadas por

los objetos de dicha clase. Sin embargo, realmente los métodos son algo más.

Si te has fijado bien, pero bien de verdad, habrás observado que las funciones `acelera()` y `frena()` definen un parámetro `self`.

```
def acelera(self):
    self.velocidad = self.velocidad + self.aceleracion
```

No obstante, cuando se usan dichas funciones no se pasa ningún argumento. ¿Qué está pasando? Pues que `acelera()` está siendo utilizada como un método por los objetos de la clase `Coche`, de tal manera que cuando un objeto referencia a dicha función, realmente pasa su propia referencia como primer argumento de la función.

NOTA: Por convención, se utiliza la palabra `self` para referenciar a la instancia actual en los métodos de una clase.

Sabiendo esto, podemos entender, por ejemplo, por qué todos los objetos de tipo `Coche` pueden referenciar a los atributos de datos `velocidad` o `color`. Son

inicializados para cada objeto en el método `__init__()`.

Del mismo modo, el siguiente ejemplo muestra dos formas diferentes y equivalentes de llamar al método `acelera()`:

```
>>> c1 = Coche('rojo', 20)
>>> c2 = Coche('azul', 20)
>>> c1.acelera()
>>> Coche.acelera(c2)
>>> print(c1.velocidad)
20
>>> print(c2.velocidad)
20
```

Para la clase `Coche`, `acelera()` es una función. Sin embargo, para los objetos de la clase `Coche`, `acelera()` es un método.

```
>>> print(Coche.acelera)
<function Coche.acelera at 0x10c60b560>
>>> print(c1.acelera)
<bound method Coche.acelera of
<__main__.Coche object at 0x10c61efd0>>
```

Atributos de clase y atributos de instancia

Una clase puede definir dos tipos diferentes de atributos de datos: *atributos de clase* y *atributos de instancia*.

Los atributos de clase son atributos compartidos por todas las instancias de esa clase.

Por el contrario, **los atributos de instancia son únicos para cada uno de los objetos** pertenecientes a dicha clase.

En el ejemplo de la clase `Coche`, `ruedas` se ha definido como un atributo de clase, mientras que `color`, `aceleracion` y `velocidad` son atributos de instancia.

Para referenciar a un atributo de clase se utiliza, generalmente, el nombre de la clase. Al modificar un atributo de este tipo, los cambios se verán reflejados en todas y cada una las instancias.

```
>>> c1 = Coche('rojo', 20)
>>> c2 = Coche('azul', 20)
>>> print(c1.color)
rojo
>>> print(c2.color)
azul
>>> print(c1.ruedas)  # At. de clase
4
>>> print(c2.ruedas)  # At. de clase
4
>>> Coche.ruedas = 6  # At. de clase
>>> print(c1.__class__.ruedas)  # At. de clase
6
>>> print(c2.ruedas)  # Atributo de clase
6
```

Como puedes observar, también se puede acceder a un atributo de clase con la nomenclatura `obj.__class__.atrib`.

Si un objeto modifica un atributo de clase, lo que realmente hace es crear un atributo de instancia con el mismo nombre que el atributo de clase.

```
>>> c1 = Coche('rojo', 20)
>>> c2 = Coche('azul', 20)
>>> print(c1.color)
rojo
>>> print(c2.color)
azul
>>> c1.ruedas = 6  # At. de instancia
>>> print(c1.ruedas)
6
>>> print(c2.ruedas)
4
>>> print(Coche.ruedas)
4
```

Herencia en Python

En programación orientada a objetos, la herencia es la capacidad de reutilizar una clase extendiendo su funcionalidad. Una clase que hereda de otra puede añadir nuevos atributos, ocultarlos, añadir nuevos métodos o redefinirlos.

En Python, podemos indicar que una clase hereda de otra de la siguiente manera:

```
class CocheVolador(Coche):
  ruedas = 6

  def __init__(self, color, aceleracion, esta_volando=False):
      super().__init__(color, aceleracion)
      self.esta_volando = esta_volando

  def vuela(self):
      self.esta_volando = True

  def aterriza(self):
      self.esta_volando = False
```

Como puedes observar, la clase `CocheVolador` hereda de la clase `Coche`. En Python, el nombre de la clase padre se indica entre paréntesis a continuación del nombre de la clase hija.

La clase `CocheVolador` redefine el atributo de clase `ruedas`, estableciendo su valor a `6` e implementa dos métodos nuevos: `vuela()` y `aterriza()`.

Fíjate ahora en la primera línea del método `__init__()`. En ella aparece la función `super()`. Esta función devuelve un objeto temporal de la superclase que permite invocar a los métodos definidos en la misma. Lo que está ocurriendo es que **se está redefiniendo el método** `__init__()` en la clase hija, usando la funcionalidad del método de la clase padre. Como la clase `Coche` es la que define los atributos

`color` y `aceleracion`, estos se pasan al constructor de la clase padre y, a continuación, se crea el atributo de instancia `esta_volando` solo para objetos de la clase `CocheVolador`.

Al utilizar la herencia, todos los atributos (atributos de datos y métodos) de la clase padre también pueden ser referenciados por objetos de las clases hijas. Al revés no ocurre lo mismo.

Veamos todo esto con un ejemplo:

```
>>> c = Coche('azul', 10)
>>> cv1 = CocheVolador('rojo', 60)
>>> print(cv1.color)
rojo
>>> print(cv1.esta_volando)
False
>>> cv1.acelera()
>>> print(cv1.velocidad)
60
>>> print(CocheVolador.ruedas)
6
>>> print(c.esta_volando)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'Coche' object has no
attribute 'esta_volando'
```

NOTA: **Cuando no se indica, toda clase hereda implícitamente de la clase `object`, de tal modo que `class MiClase` es lo mismo que `class MiClase(object)`.**

Las funciones isinstance() e issubclass()

Como ya vimos en otros temas, la función incorporada `type()` devuelve el tipo o la clase a la que pertenece un objeto. En nuestro caso, si ejecutamos `type()` pasando como argumento un objeto de clase `Coche` o un objeto de clase `CocheVolador` obtendremos lo siguiente:

```
>>> c = Coche('rojo', 20)
>>> type(c)
<class 'objetos.Coche'>
>>> cv = CocheVolador('azul', 60)
>>> type(cv)
<class 'objetos.CocheVolador'>
```

Sin embargo, Python incorpora otras dos funciones que pueden ser de utilidad cuando se quiere conocer el tipo de una clase. Son: `isinstance()` e `issubclass()`.

`isinstance(objeto, clase)` devuelve `True` si `objeto` es de la clase `clase` o de una de sus clases hijas. Por tanto, un objeto de la clase `CocheVolador` es instancia de `CocheVolador` pero también lo es de `Coche`. Sin embargo, un objeto de la clase `Coche` nunca será instancia de la clase `CocheVolador`.

`issubclass(clase, claseinfo)` comprueba la herencia de clases. Devuelve `True` en caso de que `clase` sea una subclase de `claseinfo`, `False` en caso contrario. `claseinfo` puede ser una clase o una tupla de clases.

```
>>> c = Coche('rojo', 20)
>>> cv = CocheVolador('azul', 60)
>>> isinstance(c, Coche)
True
isinstance(cv, Coche)
True
>>> isinstance(c, CocheVolador)
False
>>> isinstance(cv, CocheVolador)
True
>>> issubclass(CocheVolador, Coche)
True
>>> issubclass(Coche, CocheVolador)
False
```

Herencia múltiple en Python

Python es un lenguaje de programación que permite herencia múltiple. Esto quiere decir que una clase puede heredar de más de una clase a la vez.

```
class A:
    def print_a(self):
        print('a')

class B:
    def print_b(self):
        print('b')

class C(A, B):
    def print_c(self):
        print('c')

c = C()
c.print_a()
c.print_b()
c.print_c()
```

El script anterior dará como resultado

```
a
b
c
```

En el caso de la herencia múltiple, cuando desde una clase hija se accede a un atributo de una clase padre, ¿qué orden sigue Python para buscar entre las clases antecesoras dicho atributo?

La respuesta es que Python implementa un algoritmo conocido como *MRO* (*Method Resolution Order*) para ello. Dicho algoritmo devuelve una lista lineal del orden en el que buscar entre las clases padre. Este algoritmo es necesario dado que, siempre que se da la herencia múltiple, aparecen estructuras en forma de rombo en la jerarquía de clases.

Encapsulación: atributos privados

Encapsulación (o encapsulamiento), en programación orientada a objetos, hace referencia a la capacidad que tiene un objeto de ocultar su estado, de manera que sus datos solo se puedan modificar por medio de las operaciones (métodos) que ofrece.

Si vienes de otros lenguajes de programación, quizá te haya resultado raro que no haya mencionado nada sobre atributos *públicos* o *privados*.

Bien, por defecto, en Python, **todos los atributos de una clase** (atributos de datos y métodos) **son públicos**.

Esto quiere decir que desde un código que use la clase, se puede acceder a todos los atributos y métodos definidos en dicha clase.

No obstante, hay una forma de indicar en Python que un atributo, ya sea un dato o un método, es interno a una clase y no se debería utilizar fuera de ella. Algo así como los miembros privados de otros lenguajes. Esto es usando el carácter guion bajo `_atributo` antes del nombre del atributo que queramos ocultar.

En cualquier caso, el atributo seguirá siendo accesible desde fuera de la clase, pero el programador está indicando que es privado y no debería utilizarse porque no se sabe qué consecuencias puede tener.

También es posible usar un doble guion bajo `__atributo`. Esto hace que el identificador sea literalmente reemplazado por el texto `_Clase__atributo`, donde `Clase` es el nombre de la clase que lo define.

Un ejemplo nunca está de más.

```
class A:
    def __init__(self):
        self._contador = 0  # Privado

    def incrementa(self):
        self._contador += 1

    def cuenta(self):
        return self._contador


class B(object):
    def __init__(self):
        self.__contador = 0  # Privado

    def incrementa(self):
        self.__contador += 1

    def cuenta(self):
        return self.__contador
```

En el ejemplo anterior, la clase `A` define el atributo privado `_contador`. Un ejemplo de uso de la clase sería el siguiente:

```
>>> a = A()
>>> a.incrementa()
>>> a.incrementa()
>>> a.incrementa()
>>> print(a.cuenta())
3
>>> print(a._contador)
3
```

Como puedes observar, es posible acceder al atributo privado, aunque no se debiera.

En cambio, la clase `B` define el atributo privado `__contador` anteponiendo un doble guion bajo. El resultado de hacer el mismo experimento cambia:

```
>>> b = B()
>>> b.incrementa()
>>> b.incrementa()
>>> print(b.cuenta())
2
>>> print(b.__contador)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError:  'B'  object  has  no
attribute '__contador'
>>> print(b._B__contador)
2
```

Si te fijas, no se puede acceder al atributo `__contador` fuera de la clase. Este identificador se ha sustituido por `_B__contador`.

Polimorfismo

Polimorfismo, en programación orientada a objetos, es **la capacidad de una entidad de referenciar en tiempo de ejecución a instancias de diferentes clases**, es decir, la capacidad de tomar más de una forma.

En realidad, existen varias formas de definir Polimorfismo. Nosotros nos vamos a quedar con la anterior.

Un efecto del polimorfismo es que un método pueda presentar diferentes comportamientos en función del objeto que lo invoque. El término está muy relacionado con la herencia y la sobrecarga de métodos, aunque no siempre es así.

Es posible que este concepto te suene raro ahora mismo, pero lo vas a entender con un ejemplo. Imagina que tenemos las siguientes clases que representan animales:

```
class Perro:
    def sonido(self):
        print('Guauuuuu!!!')

class Gato:
    def sonido(self):
        print('Miaaauuuu!!!')

class Vaca:
    def sonido(self):
        print('Múuuuuuuu!!!')
```

Las tres clases implementan un método llamado `sonido()`. Ahora observa el siguiente script:

```
def a_cantar(animales):
    for animal in animales:
        animal.sonido()

if __name__ == '__main__':
    perro = Perro()
    gato = Gato()
    gato_2 = Gato()
    vaca = Vaca()
    perro_2 = Perro()
    granja = [perro, gato, vaca, gato_2,
              perro_2]
    a_cantar(granja)
```

En él se ha definido una función llamada `a_cantar()`. La variable `animal` que se crea dentro del bucle `for` es *polimórfica,* ya que, en tiempo de ejecución, hará referencia a objetos de las clases `Perro`, `Gato` y `Vaca`. Cuando se invoque al método `sonido()`, se llamará al método correspondiente de la clase a la que pertenezca cada uno de los animales.

Eliminar atributos y objetos

En reiteradas ocasiones hemos visto el uso de la sentencia `del`. Pues bien, esta sentencia también se puede usar para eliminar un atributo de un objeto del siguiente modo (seguimos con el ejemplo de la clase `Coche`):

```
>>> c1 = Coche('rojo', 20)
>>> del c1.velocidad
>>> c1.acelera()

Traceback (most recent call last):
  File "<input>", line 1, in <module>
  ...
AttributeError: 'Coche' object has no
attribute 'velocidad'
```

¿Cómo podemos eliminar un objeto en Python? Entendiendo eliminar por que dicho objeto desaparezca de la memoria.

En principio, en Python no podemos eliminar un objeto directamente. Como vimos en el tema anterior, `del x`, siendo `x` una variable que referencia a un objeto, eliminaba `x` del espacio de nombres local y su correspondiente enlace con el objeto, pero el objeto sigue estando en memoria.

Si ningún otro nombre vuelve a referenciar al objeto, este es eliminado automáticamente en un proceso conocido como *recolección de basura*.

Atributos y métodos especiales

En cualquier clase hay ciertos atributos (atributos de datos y métodos) que son considerados *especiales* y que están relacionados con la implementación de las clases en sí. En principio, Python implementa dichos atributos en las clases base del lenguaje, por lo que podemos tener acceso a ellos. Estos atributos se caracterizan porque son de la forma `__nombre__`.

Por ejemplo, en una función, `__dict__` representa el espacio de nombres local, `__name__` el nombre de la función y `__module__` el nombre del módulo en que la función es definida. Hay muchos más.

¡Cuidado si modificas los atributos de datos especiales de un objeto! Los resultados en el código pueden ser impredecibles.

En cuanto a los métodos especiales, estos se utilizan principalmente como implementaciones de ciertas operaciones que son invocadas con una sintaxis particular. Esta técnica se conoce en Python como *sobrecarga de operadores*. Sin embargo, no siempre es así, como veremos a continuación.

En los videotutoriales del tema puedes ver ejemplos de sobrecarga de operadores con implementaciones de ciertos métodos especiales.

Por ejemplo, si una clase define un método llamado `__getitem__(self, key)` y `x` es una instancia de esta clase, entonces `x[i]` sería equivalente a `type(x).__getitem__(x, i)`.

Hay tres métodos especiales que se suelen implementar cuando se define una clase personalizada. Uno de ellos ya lo hemos visto, es `__init__()`, que se utiliza para inicializar un objeto y se invoca cada vez que se crea una instancia de la clase. Los otros dos son `__str__()` y `__repr__()`.

__str__()

Este método devuelve una representación informal o amistosa en forma de cadena de caracteres de un objeto. Es llamado internamente por el constructor `str(objeto)` y las funciones `format()` y `print()`.

En caso de no implementar el método en una clase, la implementación por defecto definida en la clase `object` llama a `object.__repr__()`. Lo veremos a continuación.

__repr__()

Este método debe devolver la representación "oficial" de un objeto en forma de cadena de caracteres. Si es posible, debería verse como una expresión válida de

Python (una lista, un diccionario, etc.) que pueda usarse para recrear un objeto con el mismo valor. Si esto no es posible, se debe devolver una cadena de la forma `<... alguna descripción útil ...>`. El método es invocado internamente por la función `repr()`.

En los videotutoriales del tema puedes ver ejemplos de implementación de los métodos `__str__()` y `__repr__()`.