**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 creas una variable y le asignas un valor entero, 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 nos permite encapsular y aislar datos y operaciones que se pueden realizar sobre dichos datos.

Sigue leyendo para que entiendas qué quiero decir.

**Clases y objetos en Python**

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

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

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

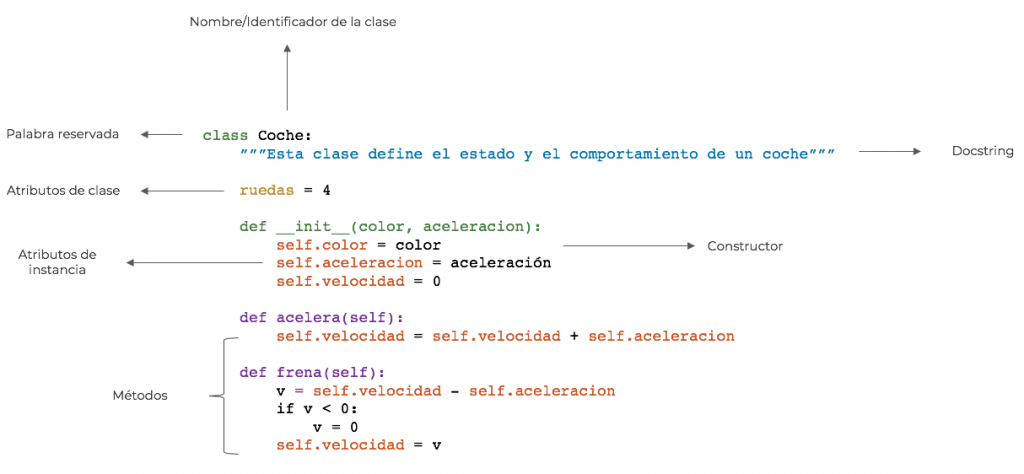
Seguro que si te digo que te imagines un coche, en tu mente comienzas a visualizar la carrocería, el color, las ruedas, el volante, si es diésel o gasolina, el color de la tapicería, si es manual o automático, si acelera o va marcha atrás, etc.

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. 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, jajaja, pero nos sirve de ejemplo). Dicha clase establece una serie datos, como ruedas, color, aceleración o velocidad y las operaciones acelera() y frena().

Cuando se crea una variable de tipo Coche, realmente se está instanciando un objeto de dicha clase. 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 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 c1 es de color rojo, mientras que 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 Python

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).

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.

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.

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 sabemos 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.

Siguiendo con nuestro ejemplo de la clase Coche, vamos a crear el siguiente objeto:

>>> 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 que, 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 parámetro 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.
* Los atributos de instancia, por el contrario, 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) # Atributo de clase

4

>>> print(c2.ruedas) # Atributo de clase

4

>>> Coche.ruedas = 6 # Atributo de clase

>>> print(c1.ruedas) # Atributo de clase

6

>>> print(c2.ruedas) # Atributo de clase

6

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 # Crea el atributo de instancia ruedas

>>> 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\_\_() de 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 Python 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 tutoriales, 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

## 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 de 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 guión 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 guión bajo \_\_atributo. Esto hace que el identificador sea literalmente reemplazado por el texto \_Clase\_\_atributo, donde Clase es el nombre de la clase actual.

Un ejemplo nunca está de más.

class A:

def \_\_init\_\_(self):

self.\_contador = 0 # Este atributo es privado

def incrementa(self):

self.\_contador += 1

def cuenta(self):

return self.\_contador

class B(object):

def \_\_init\_\_(self):

self.\_\_contador = 0 # Este atributo es 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 guión 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

Y llegamos al último concepto que veremos en este tutorial sobre programación orientada a objetos en Python.

Polimorfismo es la capacidad de una entidad de referenciar en tiempo de ejecución a instancias de diferentes clases.

Aunque este concepto te suene raro ahora mismo, 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 de la función 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.

Y bueno, esto ha sido todo en este tutorial sobre programación orientada a objetos en Python. Espero que lo hayas disfrutado y lo hayas entendido, porque es la base para ser un auténtico Pythonista