Clases y objetos

Python es un lenguaje multiparadigma, lo que significa que podemos utilizarlo enfocado a distintos paradigmas o formas de resolver problemas. En los documentos anteriores nos hemos centrado en un paradigma estructurado y modular, donde descomponíamos el código en funciones, y llamábamos a unas u otras en un orden determinado para resolver el problema.

Además de lo anterior, Python también permite estructurar nuestro código en clases, de forma que cada clase es una entidad que representa un conjunto de elementos del programa (clientes, productos, pedidos, etc.), y podremos crear objetos de esas clases que interactuarán entre sí.

1. Definición de clases y objetos

Las clases en Python se definen con la palabra `class` seguida del nombre de la clase. Después, con el código tabulado, indicamos los diferentes elementos de la clase: constructor métodos, etc. Por ejemplo, de este modo definiríamos una clase `Persona` que almacene datos básicos de personas, como su nombre, edad o teléfono:

```
class Persona:

    # Constructor
    def __init__(self, n, e, t):
        self.nombre = n
        self.edad = e
        self.telefono = t

    # Otro método de ejemplo
    def mostrar(self):
        print(self.nombre)
```

Como podemos ver, el constructor de una clase en Python se llama `__init__`. El primer parámetro que recibe (`self`) es una partícula especial que servirá para hacer referencia a cualquier elemento de la clase (atributos o métodos). De este modo, `self.nombre` hace referencia al atributo `nombre` (que no es necesario definir previamente, como en otros lenguajes). El resto de parámetros del constructor son los valores que vamos a asignar a cada atributo respectivamente: nombre (*n*), edad (*e*) y teléfono (*t*).

Adicionalmente, podemos especificar otros métodos que podamos necesitar, como en este caso el método `mostrar` que muestra por pantalla el nombre de la persona.

1.1. Instanciación de objetos

Para crear objetos de una clase, basta con usar su constructor, y para ello escribimos el nombre de la clase y le pasamos los parámetros que necesita el constructor (excluyendo el parámetro `self`, que va implícito). Así crearíamos un objeto de tipo `Persona`, y llamaríamos a su método `mostrar`:

```
p1 = Persona("María", 52, "677889900")
p1.mostrar()
```

Del mismo modo, podemos añadir objetos dentro de, por ejemplo, listas:

```
personas = [
    Persona("Juan", 70, "611223344"),
    Persona("Ana", 40, "675849301")
]
personas.append(Persona("María", 52, "677889900"))
```

1.2. Visibilidad pública o privada

Por defecto todos los elementos de una clase en Python tienen visibilidad pública. Eso quiere decir que, en el ejemplo anterior, podríamos acceder directamente a los atributos de la clase desde fuera de la misma:

```
p1 = Persona("María", 52, "677889900")
p1.edad = 53
print(p1.telefono)
```

En el caso de no querer que sea así, se deben nombrar los elementos privados anteponiéndoles el símbolo del subrayado por partida doble `__`. La clase anterior quedaría así, dejando los atributos privados:

```
class Persona:

    # Constructor
    def __init__(self, n, e, t):
        self.__nombre = n
        self.__edad = e
        self.__telefono = t

    # Otro método de ejemplo
    def mostrar(self):
        print(self.__nombre)
```

En este caso, convendría definir unos métodos públicos de acceso a los elementos privados; los típicos *getters* y *setters* de otros lenguajes, que en Python tienen una nomenclatura particular, ya que debemos utilizar ciertas anotaciones para especificar que ciertos métodos son *getters* o *setters*. Así, por ejemplo, definiríamos el *getter* y el *setter* para la edad en la clase anterior:

```
class Persona:

    # Constructor
    def __init__(self, n, e, t):
        self.__nombre = n
        self.__edad = e
        self.__telefono = t

    # Otro método de ejemplo
    def mostrar(self):
        print(self.__nombre)

    # Getter
    @property
    def edad(self):
        return self.__edad

    # Setter
    @edad.setter
    def edad(self, e):
        if e >= 0:
            self.__edad = e
```

Desde el programa principal, podemos usarlos de este modo:

```
p1 = Persona("María", 52, "677889900")
p1.edad = 30
print(p1.edad)
```

> **Ejercicio 1:**
>
> Escribe un programa en Python llamado **Jugadores.py** que defina una clase llamada `Jugador`, con atributos dorsal (numérico) y nombre (texto). Define el constructor para darles valor y un método que muestre la información de un jugador con el formato `dorsal. Nombre.`. Por ejemplo: `16. Pau Gasol`. En el programa principal, crea un par de jugadores con sus datos, y muestra su información por pantalla.

> **Ejercicio 2:**

> Escribe una nueva versión del ejercicio anterior en un programa llamado **Equipos.py** donde, además de la clase `Jugador` haya una segunda clase llamada `Equipo`, cuyo único atributo será el nombre del equipo (texto), junto con un constructor para darle valor. Haz que cada jugador pueda pertenecer a un equipo añadiendo un atributo *Equipo* a la clase *Jugador*. En el programa principal, crea un jugador y un equipo, y asigna el equipo al jugador. Muestra por pantalla la información del jugador, incluyendo el equipo.

2. Herencia

Para ilustrar el mecanismo de herencia en Python, vamos a suponer que disponemos de una clase `Persona` con un nombre y una edad como atributos (además de constructores y métodos) y de ella vamos a heredar para crear una clase `Programador`, que incorporará todos los elementos de su clase base (`Persona`) y añadirá como atributo propio el lenguaje en que programa.

- Para indicar que una clase **hereda de** otra clase, se indica el nombre de la clase a heredar entre paréntesis, tras el nombre de la clase hija.
- Para **acceder desde una clase hija a los elementos de una clase padre** se emplea el método `super()`. Se puede utilizar tanto en el constructor (para llamar al constructor de la clase padre y pasarle los parámetros que necesite) como desde cualquier otro método.
- Para **redefinir un método de la clase padre en la hija** no es necesario marcarlo de ninguna forma especidal. Simplemente se vuelve a definir el método con su nuevo código (que se puede apoyar en el del padre, si lo necesita).

Veamos todo esto en el ejemplo:

```
# Clase padre

class Persona:

    def __init__(self, n, e):
        self.nombre = n
        self.edad = e

    def mostrar(self):
        return nombre + ", " + str(edad) + " años"

    # Otros métodos...

# Clase hija

class Programador(Persona):

    def __init__(self, n, e, l):
        super().__init__(n, e)
        self.lenguaje = l

    def mostrar(self):
        return super().mostrar() + "\nPrograma en " + self.lenguaje
```

2.1. Herencia múltiple en Python

En Python se **admite herencia múltiple**, de forma que una clase puede heredar de más de una clase padre. Supongamos el caso anterior, donde la clase `Programador` hereda tanto de `Persona` como de `Empleado`. De esta última clase, incorpora el nombre de la empresa donde trabaja y el salario mensual.

```
# Clase padre 1: Persona

class Persona :
    # El código es el mismo que en el ejemplo anterior

# Clase padre 2: Empleado

class Empleado:

    def __init__(self, e, s):
        self.empresa = e
        self.salario = s

    # Otros métodos...

# Clase hija

class Programador (Persona, Empleado):

    def __init__(self, nombre, edad, empresa, salario, lenguaje):
        Persona.__init__(nombre, edad)
        Empleado.__init__(empresa, salario)
        self.lenguaje = lenguaje

    def mostrar(self):
        return self.nombre + "\nPrograma en " + self.lenguaje
```

Notar cómo, en el constructor, se antepone el nombre de cada clase delante del método al que llamar de dicha clase. En este caso, para llamar al constructor de `Persona` o de `Empleado`, con los parámetros correspondientes.

> **Ejercicio 3:**
>
> En un hospital hay diferentes tipos de personal: médicos, enfermeros y administrativos. De todos guardamos su información básica (dni, nombre, dirección y teléfono), de los médicos almacenamos también su especialidad, y de los enfermeros la planta en la que trabajan.
>
> Al hospital acuden pacientes. De todos ellos se guarda un historial con su dni, nombre, dirección, teléfono, y un conjunto de pruebas y consultas que hayan hecho en el hospital. De cada prueba o consulta guardamos la fecha en que se hizo, y el médico que le atendió
>
> Define las clases necesarias para el enunciado propuesto en un programa llamado **Hospital.py**. Define un programa principal que cree un array de personal de hospital con varios médicos y enfermeros. Define un paciente con sus datos, y dale de alta diversas pruebas realizadas por varios médicos. Finalmente, trata de mostrar por pantalla los datos completos del paciente, incluyendo su historial de pruebas.

> **AYUDA:** vídeo con la solución del ejercicio