PROGRAMACIÓN AVANZADA CON PYTHON

(CEFIRE CTEM)

Programación orientada a objetos en Python (parte 1)

Autora: María Paz Segura Valero (segura_marval@gva.es)

CONTENIDO

1. Introducción

La **Programación Orientada a Objetos** (POO) es un modelo de programación que intenta afrontar la resolución de un problema real acercándolo a la manera de entender el mundo de las personas.

Podemos observar el mundo real como un conjunto de objetos que interactúan entre sí. Por ejemplo: árboles, personas, coches, organizaciones, animales, operaciones bancarias, eventos de ocio, etc.

Cada uno de estos objetos tiene unas características que lo definen y unas acciones que se pueden llevar a cabo sobre ellos o que ellos pueden realizar por sí mismos. Por ejemplo: una persona tiene un DNI, nombre y apellidos, altura, peso, edad y puede realizar diferentes acciones como nacer, comer, beber, adoptar una mascota, criar un hijo/a, comprar o vender un coche, realizar un pago, contratar un evento de ocio, etc.

Ilustración 1.1: Pexels en Pixabay

Mediante el mecanismo de la **abstracción** podemos restringir el número de atributos y acciones que puede realizar un objeto en el mundo real para adecuarlo a la resolución del problema informático en cuestión.

Una de las ventajas de la POO es la forma de organizar los datos más cercana a la manera de entender el mundo de las personas y otra es la posibilidad de reutilizar código en situaciones similares.

En los siguientes apartados presentaremos los elementos básicos de la POO aunque ya has utilizado muchos de ellos casi sin darte cuenta porque todo en Python, absolutamente todo, son objetos. El tipo de datos **int** o **list** son un ejemplo de ello.

2. Conceptos básicos

Los elementos básicos de la *Programación orientada a objetos* son los objetos (con sus características y comportamientos) y las clases en las que se agrupan.

2.1. Clases y objetos

Un **objeto** es una entidad que tiene unas características o atributos y que ofrece unas funciones o métodos que le permiten gestionar sus atributos o interactuar con otros objetos.

Ejemplo: Individuos de distintas características físicas.

Los anteriores individuos son distintos entre sí pero todos ellos podrían agruparse en una categoría llamada *personas* con unas características y funciones comunes.

Ejemplo:

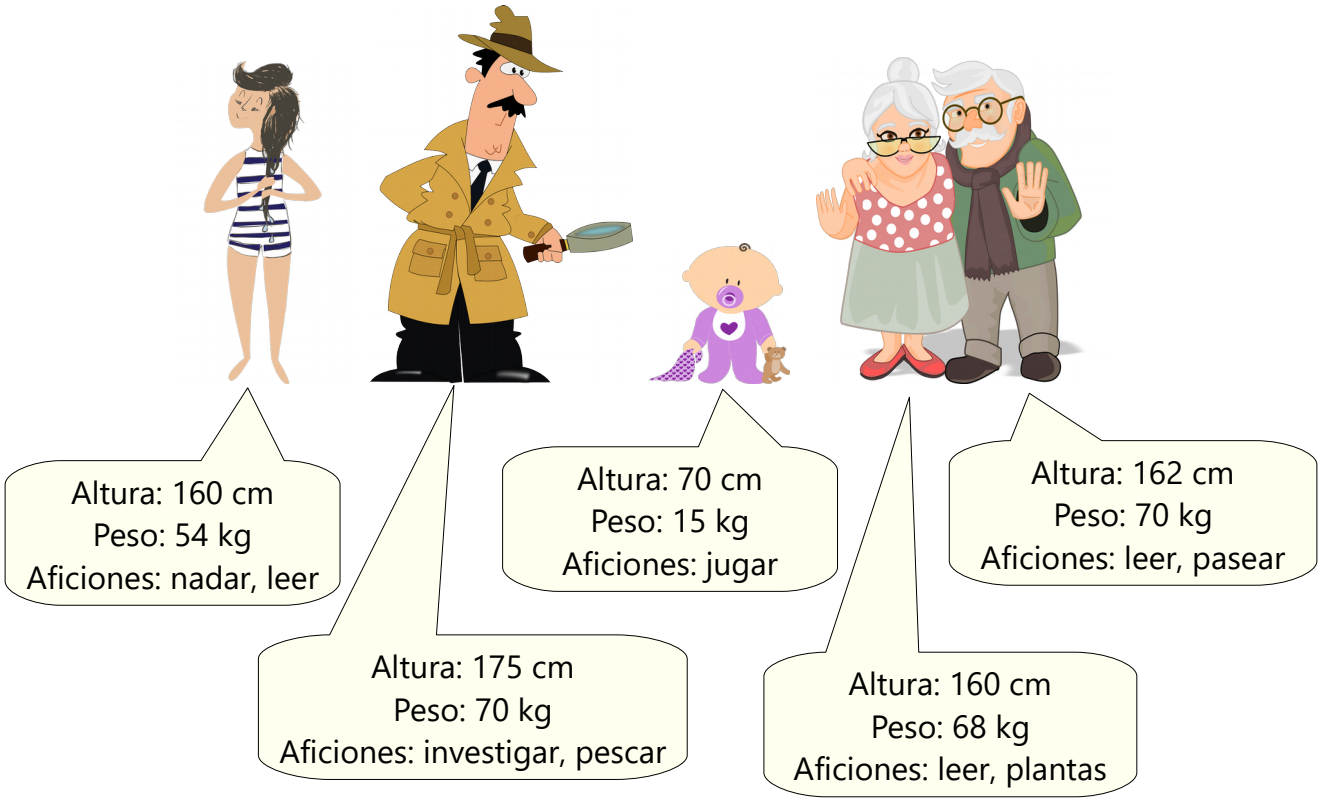
- Características: altura, peso, edad, aficiones, color del pelo, color de la piel, etc.
- Funciones: nacer, morir, alimentarse, divertirse, etc.

Todos los individuos tienen *altura* y *peso* aunque no todos ellos midan o pesen lo mismo. Y todos los individuos se alimentan y se divierten aunque no todos ellos lo hagan de la misma manera.

Así pues, podríamos considerar a los individuos como **objetos** (con sus características específicas) y a la categoría *personas* como una **clase**.

Dicho de otro modo, una **clase** podría ser la plantilla o modelo de un conjunto de objetos. En una clase definiríamos las características y funciones del conjunto de objetos y en cada uno de dichos **objetos** se concretaría el valor específico de cada característica o función.

Ejemplo: Individuos de distintas características físicas.



Haciendo una similitud con las bases de datos, podríamos decir que una clase sería una tabla y cada uno de sus registros correspondería a un objeto de la clase.

Para definir una clase en Python utilizaremos la siguiente sintaxis:

```
class nombre_clase:
  #Definición de atributos de clase
  #Definición de métodos
```

Donde ***nombre_clase*** es el nombre de la nueva clase. Por convenio, los nombres de clases empiezan por mayúsculas. Por ejemplo: Persona, Coche, Cuenta_bancaria, etc.

Cuando se define una clase en Python, se está definiendo un **nuevo tipo de datos** en el lenguaje de programación y podemos crear variables de dichos tipos. Las variables de una clase serán los objetos de la misma.

Para crear un objeto de una clase utilizaremos la siguiente sintaxis:

```
nombre_objeto = nombre_clase()
```

Donde:

- ***nombre_objeto*** será el nombre del objeto que estamos creando,
- ***nombre_clase*** será el nombre de la clase o tipo de datos que tendrá el objeto,
- entre paréntesis podemos pasar algunos parámetros. Lo veremos en el siguiente apartado.

Para conocer el tipo de datos de una variable u objeto podemos utilizar la función **type()**.

Ejemplos: Creamos varios objetos de la clase *int*.

```
>>> entero1 = int()
>>> type(entero1)
<class 'int'>
>>> entero1
0
```

```
>>> entero2 = int(20)
>>> type(entero2)
<class 'int'>
>>> entero2
20
```

2.2. Atributos y métodos

El sentido de crear clases es poder gestionar datos de un programa. Cada uno de estos datos representa una característica de un objeto de una clase determinada.

Para poder almacenar datos se crean los **atributos** de las clases. Cada dato se almacenará en un atributo que se corresponderá con una característica a gestionar.

Los valores de los atributos definen el **estado** de un objeto, es decir, las características actuales del mismo. Ejemplos de atributos: altura, peso, lista de aficiones, etc.

Por su lado los **métodos** son funciones que se definen dentro de una clase y describen el comportamiento de los objetos de la misma. Es decir, qué acciones pueden llevar a cabo. Ejemplos de métodos: nacer, morir, alimentarse, divertirse, etc.

En el siguiente ejemplo se puede ver la definición de la clase `Persona` con los atributos y métodos básicos:

```
class Persona:
    #métodos de objeto
    def __init__(self, dni, nombre): #constructor
        self.dni = dni
        self.nombre = nombre

    def imprimir(self):
        return "Dni: " + self.dni + ", Nombre: " + self.nombre
```

Podemos observar los siguientes métodos:

- `__init__()`, es el constructor de la clase (delante y detrás de la palabra `init` hay dos guiones bajos seguidos). Se trata de un método especial que se ejecuta automáticamente cuando se crea un objeto de la clase.
- `imprimir()`, es un método que se encarga de devolver una cadena de texto con el valor de los atributos del objeto.

Los dos métodos tienen un primer parámetro llamado **self**, podríamos llamarlo de otra manera, que hace referencia al objeto propietario del método que se está ejecutando. Si no lo añadiésemos, no podríamos acceder a los atributos del objeto en cuestión.

Debemos asegurarnos de que el constructor `__init__()` recibe como parámetros todos los valores necesarios para informar los atributos del objeto o el objeto se crearía sin valor en dichos atributos. En este caso recibe dos valores: el `dni` y el `nombre` de la persona.

En el cuerpo del método podemos hacer o no determinadas comprobaciones pero siempre deberemos volcar el valor de los parámetros que recibe en los atributos del objeto. Por ejemplo, en la instrucción `self.dni = dni` estamos volcando el valor del parámetro `dni` en el atributo de objeto `self.dni`.

2.2.1. Uso de una clase

Imaginemos que tenemos la clase `Persona` definida en el fichero `persona.py`. Para poder utilizar la clase, podemos hacerlo de dos formas:

a) Desde el terminal, deberemos importar la clase del módulo de la siguiente manera:

```
from persona import Persona
```

Fíjate que el nombre del paquete no incluye la extensión `.py` y que deberás estar ubicado en la carpeta donde se encuentre el fichero `persona.py`.

b) Desde Thonny, simplemente abre el fichero `persona.py` y ejecútalo.

*Al final del programa aparece una instrucción **if** un tanto peculiar. Dicha estructura sirve para poder ejecutar el módulo como un programa cualquiera y resulta útil para añadir instrucciones de prueba de la clase.*
*Cuando importemos el modulo en otro programa, la instrucción **if** no tendrá efecto.*

Para crear objetos de la clase `Persona`, lo haremos de la siguiente manera:

```
>>> p1 = Persona("111111111H", "Ana")
>>> p2 = Persona("123456789Z", "Pepa")
```

Escribimos el nombre de la clase y, entre paréntesis, el valor de sus atributos de objeto. No pasamos ningún valor para **self** porque es automático.

Podemos consultar el tipo de los dos objetos con la función `type()` que indicará que son de la clase `Persona`.

```
>>> type(p1)
<class '__main__.Persona'>
>>> type(p2)
<class '__main__.Persona'>
```

Podemos acceder a los valores de los atributos escribiendo el nombre del objeto seguido de un punto y el nombre del atributo.

```
>>> p1.dni
'11111111H'
>>> p1.nombre
'Ana'
```

De la misma manera, podemos ejecutar sus métodos:

```
>>> p1.imprimir()
'Dni: 111111111H, Nombre: Ana'
```

2.2.2. Definir valores por defecto en el constructor

Si quisiésemos tener valores por defecto en algún parámetro del constructor, podríamos indicarlos en la cabecera del método agrupados a la derecha. Es decir, los parámetros sin valor por defecto a la izquierda y los valores con valor por defecto a la derecha.

Ejemplo: Cambiamos el constructor para que `dni` sea un parámetro por defecto.

```
def __init__(self, nombre, dni = "999999999Z"):
    self.dni = dni
    self.nombre = nombre
```

En este caso podríamos crear objetos indicando el dni o no, según convenga:

```
>>> p1 = Persona("111111111H", "Ana")
>>> p2 = Persona("Pepa")
>>> p2.dni
'99999999Z'
```

2.2.3. Atributos de objeto y de clase

Hasta ahora todos los atributos que hemos utilizado son **atributos de objeto**, es decir, aquellos cuyo valor puede ser diferente en cada objeto. Estos son los más comunes.

Pero también podemos definir **atributos de clase** que son aquellos que tienen un único valor por clase y comparten todos los objetos de la misma. De hecho, los atributos de clase existen y tienen valor aunque la clase no tenga aún objetos. Sin embargo, los atributos de objeto solo tienen sentido dentro de un objeto determinado.

Ejemplo: ampliamos la definición de la clase `Persona` para añadir un *atributo de clase* llamado `idioma`. Dependiendo de su valor, los textos se escribirán en un idioma o en otro.

```
class Persona:
    #atributos de clase
    idioma = "cs"

    #métodos de objeto
    def __init__(self, dni, nombre): #constructor
        self.dni = dni
        self.nombre = nombre

    def imprimir(self):
        if Persona.idioma == "cs":
            return "Dni: " + self.dni + ", Nombre: " + self.nombre
        elif self.idioma == "va":
            return "Dni: " + self.dni + ", Nom: " + self.nombre
        else:
            return "ID Card: " + self.dni + ", Name: " + self.nombre
```

Fíjate que los *atributos de clase* se definen después del encabezado de la clase mientras que los *atributos de objeto* se definen dentro del método `__init__()`.

Además, para acceder al valor de un *atributo de clase*, podemos hacerlo a través del nombre de la clase o a través del nombre del objeto (*self* en este caso).

2.2.4. Tipos de métodos

Hasta ahora todos los atributos que hemos estado utilizando han sido **métodos de objeto** o **métodos de instancia**. Estos métodos acceden a los valores de los atributos de un objeto o instancia y, por eso, deben llamarse desde un objeto existente.

Pero puede que necesitemos crear métodos dentro de una clase que no necesiten que exista ningún objeto de la misma para funcionar. En este caso, podemos crear dos tipos de métodos:

- Los **métodos de clase** que van acompañados de la palabra (técnicamente se llama *decorador*) `@classmethod` y poseen un primer parámetro especial que suele llamarse **cls** y hace referencia a la clase que lo contiene. Es similar al parámetro **self** pero referido a la clase en lugar de al objeto.

```
@classmethod
def cambiarIdioma(cls, idioma):
    cls.idioma = idioma
```

- Los **métodos estáticos** que van acompañados del decorador `@staticmethod` y no reciben ningún parámetro especial. Suelen utilizarse cuando no necesitamos acceder a atributos de clase ni de objeto.

```
@staticmethod
def dni_es_correcto(dni):
    letras = "TRWAGMYFPDXBNJZSQVHLCKE"
    if len(dni)!=9:
        return -1
    else:
        letra = dni[8]
        num = int(dni[:8])
        if letra.upper() != letras[num % 23]:
            return -1
        else:
            return 0
```

Estos dos tipos de métodos pueden ejecutarse tanto desde la clase como desde un objeto.

```
>>> p1.cambiarIdioma("va"); p1.idioma
'va'
>>> Persona.cambiarIdioma("en"); Persona.idioma
'en'
```

```
>>> Persona.dni_es_correcto("11")
-1
>>> Persona.dni_es_correcto("11111111H")
0
```

3. Encapsulamiento

El **encapsulamiento** o **encapsulación** es un mecanismo de control que permite restringir el acceso a determinados atributos y métodos de un objeto.

De esta manera, el código externo a una clase solo podrá acceder a atributos de un objeto a través de los métodos definidos para ello pero no directamente como hemos hecho hasta el momento. Es decir, en lugar de utilizar ***nombre_objeto.nombre_atributo*** para ver el valor de dicho atributo, existirá un método que realizará las acciones necesarias para mostrarnos el valor. De esta forma no podemos manipular el atributo directamente.

En Python la ocultación de atributos es más superficial que en otros lenguajes de programación ya que, realmente, dificulta el acceso aunque no lo impide.

Para poder implementar el encapsulamiento de objetos en Python definiremos los atributos de objeto como atributos ocultos y crearemos unos métodos especiales para acceder y modificar dichos atributos. Se explica con detalle en los siguientes apartados.

3.1. Atributos ocultos

Los **atributos ocultos** llevan dos guiones bajos delante de su nombre. Por ejemplo: ***__atributo_oculto***.

Si intentamos acceder a ellos directamente, dan un error.

Ejemplo: La clase `Prueba` contiene el atributo `__oculto`.

```
>>> class Prueba():
...     def __init__(self, valor):
...             __oculto = valor
...
>>> p = Prueba(100)
>>> p.__oculto
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Prueba' object has no attribute '__oculto'
```

Entonces, ¿cómo podemos utilizarlos? Deberemos definir para cada atributo oculto un método **getter** y un método **setter**.

3.2. Métodos getter y setter

El método **getter** asociado a un atributo permite mostrar el valor de un atributo oculto.

Sintaxis:

```
@property
def nombre_atributo(self):
    return self.__nombre_atributo
```

Ejemplo:

```
@property
def dni(self):
    return self.__dni
```

Igual en un ejemplo tan sencillo no se le ve mucha utilidad pero, por ejemplo, podríamos aprovechar el **getter** para realizar la conversión de la altura de una persona a la unidad de medida establecida en un atributo de clase en lugar de mostrarla directamente como estuviese almacenada en el objeto.

Para poder volcar un valor en el atributo oculto deberemos utilizar el método **setter**, cuya sintaxis se muestra a continuación:

```
@nombre_atributo.setter
def nombre_atributo(self, valor):
    self.__ nombre_atributo =  valor
```

Ejemplo: Validamos el `dni` pasado por parámetro con el método estático `dni_es_correcto()`. Si está bien, lo guardamos en el atributo oculto `dni` y, si no, lo dejamos en blanco y lanzamos una excepción con la instrucción **raise**.

```
@dni.setter
def dni(self, dni):
    if (self.dni_es_correcto(dni) == 0):
        self.__dni = dni
    else:
        self.__dni = ""
        raise ValueError("Formato incorrecto del DNI")
```

La instrucción **raise** permite lanzar una excepción en el programa del tipo que queramos. Como parámetro se pasa el mensaje que deseemos mostrar al usuario.

Aquí podemos ver la creación de dos personas: una con el `dni` correcto y otra con el `dni` incorrecto.

```
>>> p1 = Persona("11111111H", "Ana")
>>> p1.imprimir()
'Dni: 11111111H, Nombre: Ana'
>>> p1 = Persona("1H", "Pepe")
>>> p1.imprimir()
'Dni: , Nombre: Pepe'
```

Además, en el constructor de la clase utilizaremos los métodos **setter** directamente en lugar de los atributos ocultos. Así, ellos serán los encargados de realizar las validaciones pertinentes y los cambios en los valores de los atributos ocultos.

Ejemplo:

```
#métodos de objeto
def __init__(self, dni, nombre): #constructor
    self.dni = dni
    self.nombre = nombre
```

3.3. Reescritura de métodos

Es habitual reescribir determinados métodos en las nuevas clases para añadir funcionalidad extra a los ya existentes.

El constructor `__init__()` es uno de los métodos que hemos reescrito en todas nuestras clases. Pero también es habitual reescribir los siguientes:

- `__str__()` para devolver una cadena de texto cuando se intenta convertir el objeto a una cadena de texto o se utiliza la función `print()`.

```
def __str__(self):
    return ("(" + self.dni + ", " + self.nombre + ")")
```

```
>>> p1 = Persona("11111111H", "Ana")
>>> print(p1)
 (11111111H, Ana)
>>> str(p1)
'(11111111H, Ana)'
```

- `__eq__()` para poder comparar dos objetos de la misma clase y saber si contienen la misma información. Si no lo reescribimos, al ejecutar `objeto1 == objeto2`, será como ejecutar `id(objeto1) == id(objeto2)`.

- Ejemplo 1: antes de reescribir el método `__eq__()`.

```
>>> p1 = Persona("11111111H", "Ana")
>>> p2 = Persona("11111111H", "Ana")
>>> p1 == p2
False
```

Ejemplo 2: después de reescribir el método `__eq__()`.

```
def __eq__(self, otro):
    return ((self.dni == otro.dni) and (self.nombre == otro.nombre))
```

```
>>> %Run persona_v5.py
>>> p1 = Persona("11111111H", "Ana")
>>> p2 = Persona("11111111H", "Ana")
>>> p1 == p2
True
>>> id(p1) == id(p2)
False
```

En la siguiente página de Python se puede ver un listado de ellos: https://docs.python.org/3.4/reference/datamodel.html#special-method-names

4. Fuentes de información

- Programación orientada a objetos de *José Domingo Muñoz*: https://plataforma.josedomingo.org/pledin/cursos/python3/curso/u50/
- Programación orientada a objetos (POO) en Python de *Juan José Lozano Gómez*: https://j2logo.com/python/tutorial/programacion-orientada-a-objetos/
- Programación orientada a objetos: https://entrenamiento-python-basico.readthedocs.io/es/latest/leccion9/poo.html