Sesión 4 - Pramación orientada a objetos.

Programación orientada a objetos.

Con Programación Orientada a Objetos (POO) nos referimos al paradigma de programación que se enfrenta a los problemas creado una representación de todos los elementos que aparecen en él así como de las relaciones entre ellos.

Cuando un problema es complejo o simplemente muy extenso para manejar todos los datos en un ámbito global, la programación orientada a objetos es útil porque crea capas aisladas de abstracción que pueden interaccionar entre ellas. Esta ventaja que ofrece la POO es en realidad de cara al usuario, nunca al programa.

Definición en Python.

Hasta ahora en el curso hemos hemos visto diferentes tipos de datos. Estos tipos podían ser simples (enteros, strings) o estructurados (listas, tuplas, diccionarios, etc.). Además, hemos visto que cada tipo de dato traía consigo una serie de métodos (funciones) propias.

Las **clases** amplían la definición de tipo, permitiendo definir nuestros propios objetos con métodos y atributos propios. Definamos un poco estos conceptos:

- Llamamos **clase** al conjunto de datos, métodos y propiedades que abstraen o representan un concepto.
- Llamamos **objeto** a toda instancia creada a partir de una clase.
- Llamamos **atributos** a los datos definidos en las clases y que son accesibles desde los objetos.

- Llamamos **método** a las funciones definidas en las clases.
- Llamamos **instancia** a la especificación de una clase, es decir, a un objeto creado a partir de una clase con unas condiciones particulares. Es un anglicismo que podemos sustituir por 'modelo' o 'ejemplo'.

Una forma de entender la POO es considerar a la clase como una **plantilla** y a los objetos como elementos creados a través de ella.

> Recuerda: En Python todos las variables son instancias de alguna clase, es decir, todo son objetos.

Para saber si un objeto es una instancia de alguna clase se utiliza la función `isinstance`.

```
x = 5
isinstance(x, int)  # True
isinstance(x, str)  # False
isinstance(x, object)  # Siempre True
```

Para comprobar que dos variables hacen referencia a la misma instancia tenemos los operadores `is` e `is not`:

```
x = [1]
y = x
z = [1]
print(y is x)  # True
print(z is x)  # False
```

Definir clases.

Para definir nuestras clases utilizamos la palabra `class`.

```
class Perro:
    'Representa un perro.'  # Comentario
    pass
```

Podemos añadir atributos a la clase **definiéndolos** (puedes declararlos pero no tendrá efectos) dentro del bloque.

```
class Perro:
    'Representa un perro.'
    edad = 0
```

Para conseguir una instancia de una clase (un objeto), podemos crearla a partir del **constructor de clase** utilizando el nombre de la clase y el operador `()`.

```
class Perro:
    'Representa un perro.'
    edad = 0
zeus = Perro()
isinstance(zeus, Perro)  # True
```

Ahora podemos modificar los atributos del objeto. En Python también se nos permite añadir atributos nuevos a objetos ya creados sin dejar de ser estos instancias de su clase. Esto no es recomendable, pues no tendríamos información de los atributos que tiene un objeto.

```
zeus.edad = 3
zeus.nombre = 'Zeus'  # Vale, pero está feo.
print(f'Mi perro {zeus.nombre} tiene {zeus.edad} años.')
```

Las funciones que definamos dentro de la clase pasarán a ser métodos de la misma. Cuando llamamos a un método desde un objeto, **estamos pasando como primer argumento el propio objeto**. Para hacer referencia a una instancia concreta y acceder a sus métodos y atributos dentro de la definición de una clase utilizamos la palabra `self`.

```
class Perro:
    'Representa un perro.'
    edad = 0
    nombre = ''

    def saluda(self):
        'Saludo (y despedida) de un perro'
        print('Guau!')

    def info(self):
        'Devuelve información de un perro.'
        print(f'Nombre: {self.nombre}', f'Edad: {self.edad}', sep='\n')

zeus = Perro()
zeus.nombre, zeus.edad = 'Zeus', 3
zeus.saluda()
zeus.info()
```

Podemos utilizar los métodos desde la clase, pero le tenemos que pasar como argumento una instancia.

```
Perro.info()  # Error!
Perro.info(zeus)  # Igual que zeus.info()
```

Métodos mágicos.

Existen una serie de métodos especiales (llamados **mágicos**) que definen funcionalidades de instancias de una clase que no pueden ser representadas en un método regular. Estos métodos están definidos entre barras bajas `__nombre_metodo__`.

Existen muchos métodos mágicos (eventualmente añaden algunos en nuevas versiones). Veremos ahora los más importantes y a medida que se vean nuevas funcionalidades del lenguaje se revelarán nuevos.

Constructor.

El método `__init__` se utiliza como **constructor** de una clase. Un constructor es un método que configura una instancia cuando es creada. No tiene retorno y se suele utilizar para introducir las variables que necesita una clase para especializarse.

```
class Perro:
    'Representa un perro.'
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def saluda(self):
        'Saludo (y despedida) de un perro'
        print('Guau!')

    def info(self):
        'Devuelve información de un perro.'
        print(f'Nombre: {self.nombre}', f'Edad: {self.edad}', sep='\n')

dogo = Perro('Cookie', 4)  # Pasamos el nombre y la edad.
```

> En Python no hay sobrecarga de funciones o métodos. Eso significa que no pueden haber dos constructores. Si queremos instanciar una clase de diferentes formas tenemos que discriminar la entrada manualmente.

> No confundir el método `__init__` con el método `__new__`. Este último no tiene que ver con la instancia sino con la clase.

Personalización básica.

Algunos métodos de personalización básica son:

- `__repr__(self)`: Retorna un string "oficial" de representación de un objeto. Debería verse, si es posible, como una expresión de Python válida que puede ser utilizada para recrear un objeto con el mismo valor o, en su defecto, una representación útil de este.
- `__str__(self)`: Crea una representación de texto de un objeto para mostrar. Es la representación que obtenemos con `str.format` y `print`. Si no se define, se usa `__repr__`.
- `__format__(self)`: Especializa todavía más `__str__` la representación de `str.format`.
- `__bool__(self)`: Define como se evalúa en objeto en una expresión condicional, en un `if`, en la conversión a `bool`, etc.
- `__call__`: Permite utilizar la instancia como una función *callable*.

```
class Perro:
    'Representa un perro.'
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def __call__(self):
        'Saludo (y despedida) de un perro'
        print('Guau!')

    def __str__(self):
        'Representación del perro'
        return f'Nombre: {self.nombre} - Edad: {self.edad}'

dogo = Perro('Cookie', 4)  # Pasamos el nombre y la edad.
print(dogo)
dogo()
```

Operadores.

Existen métodos mágicos para manejar todos los operadores. Una lista de los básicos:

- `__add__(self, other)` (+)
- `__sub__(self, other)` (-)
- `__mul__(self, other)` (*)
- `__truediv__(self, other)` (/)
- `__floordiv__(self, other)` (//)
- `__mod__(self, other)` (%)
- `__pow__(self, other)` (**)
- `__and__(self, other)` (and)
- `__or__(self, other)` (or)
- `__lt__(self, other)` (<)
- `__le__(self, other)` (<=)
- `__eq__(self, other)` (==)
- `__ne__(self, other)` (!=)
- `__gt__(self, other)` (>)
- `__ge__(self, other)` (>=)

Por ejemplo, una clase que utiliza el operador suma:

```
class CestaFrutas:
    'Representa una cesta de fruta'
    def __init__(self, *frutas):
        self.cesta = {}
        for fruta in frutas:
            if fruta in self.cesta:
                self.cesta[fruta] += 1
            else:
```

```
                self.cesta[fruta] = 1

    def muestra_cesta(self):
        'Imprime la cesta de frutas'
        print(self.cesta)

    def __add__(self, otra_cesta):
        'Suma dos cestas'
        if not isinstance(otra_cesta, CestaFrutas):
            raise ValueError('Solo se pueden sumar dos cestas!')
        salida = CestaFrutas()
        for fruta in self.cesta:
            if fruta in salida.cesta:
                salida.cesta[fruta] += self.cesta[fruta]
            else:
                salida.cesta[fruta] = self.cesta[fruta]
        for fruta in otra_cesta.cesta:
            if fruta in salida.cesta:
                salida.cesta[fruta] += otra_cesta.cesta[fruta]
            else:
                salida.cesta[fruta] = otra_cesta.cesta[fruta]
        return salida

print('cesta_1:')
cesta_1 = CestaFrutas('Pera', 'Manzana', 'Mango', 'Pera', 'Pera')
cesta_1.muestra_cesta()
print('cesta_2:')
cesta_2 =CestaFrutas('Plátano', 'Pera', 'Mango', 'Pera', 'Pera')
cesta_2.muestra_cesta()
print('cesta grande:')
cesta_grande = cesta_1 + cesta_2
cesta_grande.muestra_cesta()
```

> La función `sorted` funciona definiendo los métodos `__lt__` y `__eq__`.

Existen más operadores, puedes ver la lista completa en la documentación.

Simulando contenedores.

Se puede hacer que una clase emule a un contenedor (lista, diccionario, etc.). Para ello existen los siguientes métodos:

- `__len__(self)`: Se llama cuando se pasa la instancia por la función `len()` y devuelve la "longitud" o "tamaño" del objeto.
- `__contains__(self, obj)`: Permite utilizar el operador de pertencencia `in`.
- `__getitem__(self, key)`: Permite utilizar el operador de acceso `[]` para obtener un valor.
- `__setitem__(self, key, value)`: Permite utilizar el operador de acceso `[]` para introducir un valor.
- `__delitem__(self, key)`: Se llama cuando se ejecuta `del obj[key]`.

Un ejemplo sencillo:

```
class Matriz:
    "Representa una matriz"

    def __init__(self, m, n):
        self.m = m # Filas
        self.n = n # Columnas
        self.data = [[0]*n for _ in range(m)]

    def __getitem__(self, key):
        return self.data[key]

    def __setitem__(self, key, value):
        "Se llama cuando ponemos una fila entera."
        self.data[key] = value
```

En la documentación puedes encontrar una descripción más detallada de estos y otros métodos para contendedores.

Ejercicio de clase.

Crea un módulo con la clase `CuentaBancaria` que tenga los métodos básicos para ver el saldo, ingresar, retirar y transferir dinero a otra cuenta. Después crea dos instancias en un script aparte y realiza operaciones de ingreso, retirada y transferencia.

Métodos de clase.

Los métodos de clase son métodos que se usan para realizar operaciones en el ámbito de clase y no en las instancias de esta. Se definen mediante el decorador `@classmethod` y a diferencia de los métodos normales, se les pasa como primer argumento el identificador `cls` (en vez de `self`) como representación de los atributos de la clase.

```
class Gato:
    "Representa un gato"
    peso_kg = 2  # Peso medio

    def __init__(self, nombre, peso = None):
        self.nombre = nombre
        if peso is not None:
            self.peso_kg = peso

    @classmethod
    def peso_medio(cls):
        return cls.peso_kg

print(f'El peso medio de un gato es de {Gato.peso_medio()} kg.')
```

Se suelen utilizar como una colección de utilidades relacionadas con la clase, para obtener datos genéricos o para generar instancias de la clase de forma especial (esta es la más interesante quizá).

```
class Biblioteca:
    'Gestión de una biblioteca'

    def __init__(self, dicc_libros):
        'Normalmente se inicializa con un diccionario de libros.'
        ...

    @classmethod
    def from_json(cls, ruta):
        'Creamos una instancia desde los datos de un archivo json.'
        with open(ruta, 'r') as archivo_json:
            libros = json.load(libros)
        return cls(libros)  # Devuelve una instancia de la clase.
```

Métodos estáticos.

Podemos definir métodos dentro del ámbito de una clase que no estén asociados a ninguna instancia particular ni a los atributos de la clase. Estos métodos se llaman estáticos se definen como funciones normales dentro de la clase con el decorador `@staticmethod`.

```
class Pizza:
    "Representa una Pizza"
    def __init__(self, toppings):
        self.toppings = toppings

    @staticmethod
    def validar(topping):
        "Valida un topping según el Código Penal Italiano."
        if topping == "piña":
            print("¡MAMMA MIA!")
            return False
        else:
            return True

ingredientes = ['queso', 'cebolla', 'jamón', 'piña']
pizza = Pizza([topping for topping in ingredientes if
Pizza.validar(topping)])
```

Se utilizan con funciones que tienen que ver con el ámbito abstracto de la clase pero no interaccionan directamente con ella o con sus instancias.

Propiedades.

En Python tienes por defecto acceso libre a todos los atributos de una instancia. Esto puede ser problemático en APIs públicas donde los usuario cambien parámetros importantes para el funcionamiento de la instancia.

En C++, por ejemplo, para administrar el acceso a los atributos tienen un sistema de atributos públicos y privados. Python por su parte cuenta con el decorador `@property` que define el acceso que se tiene a un atributo (solo lectura, lectura-escritura, borrado...).

La forma 'tradicional' de acceso a atributos en otros lenguajes pasa por definir *getters* y *setters* para estos.

```
class Punto:
    "Define un punto"
    def __init__(self, x, y):
        self._x = x  # Una variable con '_' es para indicar que es oculta.
        self._y = y

    def get_x(self):
        "Devuelve la coordenada x"
        return self._x

    def set_x(self, value):
        "Introduce valor en la coordenada x"
        self._x = value

    def get_y(self):
        "Devuelve la coordenada y"
        return self._y

    def set_y(self, value):
        "Introduce valor en la coordenada y"
        self._y = value

punto = Punto(1, 2)
punto.set_x(4)
punto.set_y(3)
print(f'Punto en ({punto.get_x()},{punto.get_y()})')
```

En este código se pretende que los atributos `_x` y `_y` no sean accesibles y se definen métodos de acceso *getters* (para obtener valores) y *setters* (para escribir). Veamos una solución con `@property`:

```
class Punto:
    "Define un punto"
    def __init__(self, x, y):
        self._x = x
        self._y = y

    @property
    def x(self):
        "Coordenada x"
        return self._x

    @x.setter
    def x(self, valor):
        self._x = valor
```

```
    @property
    def y(self):
        "Coordenada y"
        return self._y

    @y.setter
    def y(self, valor):
        self._y = valor

punto = Punto(1, 2)
punto.x = 4
punto.y = 3
print(f'Punto en ({punto.x},{punto.y})')
```

La interfaz de acceso a los atributos ahora es más plana y directa, evitando la llamada a un método con `()`. `x` e `y` se comportan como atributos pero tienen asociado un método. A este tipo de atributos los llamamos **propiedades**.

Funcionan del siguiente modo:

- Se decora una función con el nombre de la propiedad.
- El decorador ha creado un *getter* y ha generado en el ámbito de la clase otro decorador con el nombre de la propiedad.
- Con dicho decordador ahora podemos crear el **setter** que permite cambiar el valor y el **deletter** que define como borrarlo (no lo vamos a ver).

Control de acceso.

Las propiedades están a medio camino entre métodos y atributos. Gracias a eso podemos controlar el acceso a estas.

Propiedades de solo lectura.

Si queremos crear propiedades de solo lectura **no definimos el setter**.

```
class PuntoFijo:
    "Define un punto"
    def __init__(self, x, y):
        self._x = x
        self._y = y

    @property
    def x(self):
        "Coordenada x"
        return self._x

    @property
    def y(self):
        "Coordenada y"
```

```
        return self._y

punto = PuntoFijo(1, 2)
print(f'Punto en ({punto.x},{punto.y})')
punto.x = 4  # AtributeError!
punto.y = 3
```

Propiedades de solo escritura.

Aunque es raro, puede ocurrir que se quiera introducir una variable de solo lectura. Para ello **se debe lanzar un error en el getter**.

```
class Usuario:
    "Define un punto"
    def __init__(self, login, password):
        self.login = login
        self.password = password

    @property
    def password(self):
        "Contraseña"
        raise AttributeError('Error! No puedes ver la contraseña!')

    @password.setter
    def password(self, password):
        self._password = Usuario.encriptar(password)

    @staticmethod
    def encriptar(password):
        'Debería encriptar la contraseña'
        return password + ' - encriptada'

usuario = Usuario('pepe98', 'swiftie4ever')
print(usuario.password)
```

Propiedades de lectura y escritura.

Se puede generan propiedades de lectura y escritura como en el primer ejemplo. Para ello **se defienen tanto el getter como el setter**.

Propiedades calculadas.

Gracias a que tenemos un acceso de tipo atributo pero se ejecuta un método, podemos definir "atributos" que no existen realmente sino que son calculados al acceder a la propiedad.

```
import math

class Circulo:
```

```
    "Define una círculo en el espacio"

    def __init__(self, x, y, radio):
        self.x = x
        self.y = y
        self.radio = radio

    @property
    def radio(self):
        "Radio del círculo"
        return self._radio

    @radio.setter
    def radio(self, nuevo_radio):
        if nuevo_radio <= 0:
            raise ValueError('¡El radio debe ser mayor que 0!')
        self._radio = nuevo_radio

    @property
    def diametro(self):
        "Diámetro del círculo."
        return 2*self._radio

    @property
    def area(self):
        "Área del círculo."
        return math.pi*self._radio**2

circulo = Circulo(0, 0, 3)
print('Diámetro:', circulo.diametro)
print('Área:', circulo.area)
```

Iteradores.

Con los conocimientos adquiridos es hora de entender como funciona la iteración en Python.

Cuando iteramos sobre un objeto, por ejemplo en un `for`, lo que hacemos es pasarlo al principio por la función `iter`, que nos va a devolver un objeto **iterador**. El objeto iterador es cualquiera que tiene definido el método `__next__(self)`, el cual se encarga de proveer valores mientras no se lance una excepción `StopIteration` que para el bucle.

> En los generadores la excepción `StopIteration` se lanza sola al alcanzar un `return`. **NO se deben lanzar excepciones `StopIteration` en generadores** pues provocaría un error no manejable.

La función `next(iterador)` devuelve el siguiente elemento de un iterador (llama a `__next__`). Veamos un ejemplo con una lista:

```
data = [1, 2, 3]
data_iter = iter(data)
print(next(data_iter))
print(next(data_iter))
```

```
print(next(data_iter))
print(next(data_iter))  # Error!
```

La clase `list` tiene definido un método `__iter__(self)` que devuelve un interador con los valores de la lista. El iterador nos va devolviendo valores hasta que se queda sin elementos sobre los que iterar y devuelve la excepción `StopIteration`.

Por resumir definiciones:

- **Iterable**: todo objeto sobre el que se puede iterar (tiene definido `__iter__(self)`). Es el argumento que acepcta `iter(iterable)`.
- **Iterador**: objeto sobre el que se itera (tiene definido `__next__(self)`).
- **Generador**: función especial que funciona como iterador.

Iteradores puros.

Ahora vamos a definir nuestros propios iteradores. Un iterador puro sería aquel sobre el que se puede iterar directamente. Veamos una definición típica.

```
class IteradorPrimos:
    "Iterador que devuelve los n primeros números primos"

    def __init__(self, n_numeros):
        self.n = n_numeros
        self.ultimo = 1

    def __next__(self):
        if self.n == 0:
            raise StopIteration
        self.n -= 1
        while True:
            self.ultimo += 1
            if self.es_primo(self.ultimo):
                return self.ultimo

    @staticmethod
    def es_primo(candidato):
        "Comprueba si el número es primo."
        for i in range(2, candidato//2):
            if (candidato % i == 0):
                return False
        return True

primos = IteradorPrimos(5)
print(next(primos))
print(next(primos))
print(next(primos))
print(next(primos))
print(next(primos))
print(next(primos))  # ¡StopIteration!
```

Si intentamos introducir un iterador en un `for` directamente nos saltará un error pues **no es un objeto iterable**, sino un iterador. Para usarlo en un `for` debemos integrarlo en una clase **iterable**.

```
class Primos:
    "Clase que genera primos"

    def __init__(self, n_numeros):
        self.n = n_numeros

    def __iter__(self):
        return IteradorPrimos(self.n)

for i in Primos(10):
    print(i)
```

La secuencia que se sigue en el `for` es equivalente a:

```
iterable = Primos(10)
iterador = iter(iterable) # de tipo IteradorPrimos
while True:
    try:
        i = next(iterador)
    except StopIteration:
        break
    else:
        print(i)
```

Iterador iterable.

Si se integran ambos métodos en una clase nos encontramos con un iterador iterable:

```
class Primos:
    "Iterador iterable que devuelve los n primeros números primos"

    def __init__(self, n_numeros):
        self.n = n_numeros
        self.ultimo = 1

    def __iter__(self):
        return self

    def __next__(self):
        if self.n == 0:
            raise StopIteration
        self.n -= 1
        while True:
            self.ultimo += 1
```

```
            if self.es_primo(self.ultimo):
                return self.ultimo

    @staticmethod
    def es_primo(candidato):
        "Comprueba si el número es primo."
        for i in range(2, candidato//2):
            if (candidato % i == 0):
                return False
        return True

for i in Primos(10):
    print(i)
```

Gestión de contexto.

Un contexto es un espacio del programa donde hay disponibles ciertos recursos (funciones, objetos, APIs, etc.). Los gestores de contexto permiten asignar y liberar esos recursos a demanda. En Python la etiqueta `with` permite crear contextos. La sintaxis es:

```
with gestor_contexto() [as VAR]:
    # context
```

Un ejemplo de contexto es el que usamos para leer/escribir archivos:

```
with open('archivo.txt', 'r') as archivo:
    data = archivo.read()
```

El equivalente de este código es:

```
archivo = open('archivo.txt', 'r')
try:
    data = archivo.read()
finally:
    archivo.close()
```

El gestor de contexto se ha ocupado de abrir el archivo y de cerrarlo. En Python la forma más sencilla de crear un operador de contexto es con los métodos `__enter__(self)` y `__exit__(self, *error_handler)`. Veamos cómo implementar el ejemplo de `open` nosotros mismos.

```
class Archivo(object):
    "Gestiona la operación de abrir/cerrar archivo"

    def __init__(self, ruta, metodo = 'r'):
```

```
        self.archivo = open(ruta, metodo)
    def __enter__(self):
        return self.archivo
    def __exit__(self, *info_error):
        self.archivo.close()
```

Con este gestor de contexto abrir archivos sin peligro de dejarlos abiertos si hay algún problema.

```
with Archivo('demo.txt', 'w') as archivo:
    datos = archivo.read()  # Error, el archivo es de solo escritura
```

Aunque se lanza una excepción el archivo se cierra. Podemos también gestionar la excepción de forma silenciosa desde el método `__exit__` si devolvemos un `True`:

```
class Archivo(object):
    "Gestiona la operación de abrir/cerrar archivo"

    def __init__(self, ruta, metodo = 'r'):
        self.archivo = open(ruta, metodo)
    def __enter__(self):
        return self.archivo
    def __exit__(self, tipo_error, valor_error, traza_error):
        if tipo_error is not None:
            print('Ha ocurrido una excepción:', valor_error)
        self.archivo.close()
        return True

with Archivo('demo.txt', 'w') as archivo:
    datos = archivo.read()
```

Ahora mismo no es muy útil, pero más adelante nos ayudará a gestionar conexiones.

Ejercicio de clase.

Crea un módulo `logistica` con, al menos, las siguientes clases:

- `Pallet`: unidad de transporte con los atributos `carga`, `peso`, `uuid` (identificador único).
- `Pedido`: abstraccion de un envío con los atributos `pallets`, `destino`, `cliente`, `uuid`.
- `Camion`: abstracción de un camión con el atributos de `carga_maxima`, `pedido`, `ruta`.

Añade las funciones y métodos que sean necesarios para que:

- Puedas añadir **pallets** a un **pedido**.
- Puedas asignar **pedidos** a un **camión**.
- La ruta del camión se defina a partir de los pedidos.
- El camión no pueda sobrepasar el peso máximo.
- Puedas leer una lista de pedidos de un JSON.

Crea otro script que importe el módulo `logistica` y que, a partir de los pedidos del archivo `pedidos.json`, imprima por pantalla un informe de los camiones que son necesarios para llevar los paquetes con su ruta y pedidos.

> Define la carga máxima de los camiones del ejemplo en `18.000` kg.